Hot Swapping Protocol Implementations in the OPNET Modeler Development Environment

THESIS

Mark E Coyne, 2d Lt, USAF

AFIT/GCS/ENG/08-05

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT/GCS/ENG/08-05

Hot Swapping Protocol Implementations in the OPNET Modeler Development Environment

THESIS

Presented to the Faculty

Department of Electrical and Computer Engineering

Graduate School of Engineering and Management

Air Force Institute of Technology

Air University

Air Education and Training Command

In Partial Fulfillment of the Requirements for the

Degree of Master of Science (Computer Science)

Mark E Coyne, BS

2d Lt, USAF

March 2008

AFIT/GCS/ENG/08-05

Hot Swapping Protocol Implementations in the OPNET Modeler Development Environment

Mark E Coyne, BS
2d Lt, USAF

Approved:

/signed/	7 Feb 2008
Maj. Scott Graham, PhD (Chairman)	Date
/signed/	7 Feb 2008
Lt. Col. Stuart Kurkowski, PhD (Member)	Date
/signed/	7 Feb 2008
Dr. Kenneth Hopkinson, PhD (Member)	Date

AFIT/GCS/ENG/08-05

Abstract

This research effort demonstrates hot swapping protocol implementations in OPNET via the building of a dependency injection testing framework. The thesis demonstrates the externalization (compiling as stand-alone code) of OPNET process models, and their inclusion into custom DLL's (Dynamically Linked Libraries). A framework then utilizes these process model DLL's, to specify, or "inject," process implementations post-compile time into an OPNET simulation. Two separate applications demonstrate this mechanism. The first application is a toolkit that allows for the testing of multiple routing related protocols in various combinations without code re-compilation or scenario re-generation. The toolkit produced similar results as the same simulation generated manually with OPNET. The second application demonstrates the viability of a unit testing mechanism for the externalized process models. The unit testing mechanism was demonstrated by integrating with CxxTest and executing xUnit style test suits.

Acknowledgements

I would like to thank Lt Col Timothy Halloran, Lt Col Jeffrey McDonald, and Lt Col Stuart Kurkowski. As the professors of the software engineering track, they have prepared me well for this thesis and the many future software related endeavors to come. I would also like to thank Dr. Kenneth Hopkinson for keeping me on track and continually providing guidance and support. Lastly, I would like to extend a special thanks to my advisor, Maj Scott Graham, for the many invaluable thesis and Air Force discussions, his willingness to go the extra mile to help his advisees, and his genuine ability to inspire those around him to seek knowledge.

Mark E Coyne

Table of Contents

List of Figures

Figure Page

List of Tables

List of Listings

Hot Swapping Protocol Implementations in the OPNET Modeler Development Environment

I. Introduction

Software toolkits provide researchers with the background and support they need to aid them in their area of study. *Design Patterns* defines toolkits as "A collection of classes that provides useful functionality but does not define the design of an application [15]." In the network simulation research community, there exist a demand for a network simulation toolkit that allows for the design and specification of a simulation with key modules of network objects such as routers, easily hot swappable and re-configurable. Duggan describes "hot swapping" as "the ability to change or "swap" the module [of] implementation [at run time] without the client threads noticing the change [7]." With the ability to hot swap individual modules of a network object's implementation, researchers can better study their interaction and effect on the overall network.

Several toolkits are currently available to the researcher that provide similar functionality. The Click Modular Router [18] is a modular, hot swappable router implementation architecture. The network research and development community has borrowed from the modular, fine grained interface design of Click with the development of the discrete event simulator called OPNET Modeler. However, while the OPNET Modeler development effort maintained the modular architecture of Click, the ability to hot swap implementations was lost. This research seeks to restore this ability in the OPNET Modeler simulation environment and provide a standardized application programming interface (API) for the specification, execution, and data collection of OPNET simulations.

This research is a part of two larger research efforts. The first seeks to build a similar toolkit that provides simulator independence in addition to hot swapping

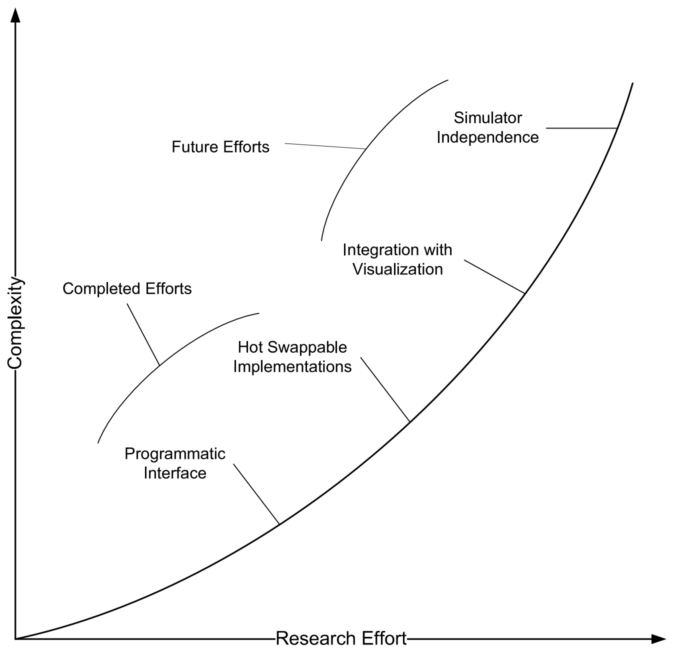

Figure 1.1: Long-Term Toolkit Development Plan. The ultimate goal of the toolkit is to provide simulator independence and integration ability with a network visualizer in addition to API's for the specification, execution, and data collection of OPNET simulations.

ability and programmatic control. The second major effort seeks to incorporate the toolkits together with visualization software in an integrated framework. Development of the programmatic interface and hot swappable network object implementations are covered in this research. The software this thesis describes contains rudimentary API's for Java network visualization integration, but was not extensively tested or developed. Additionally, simulator independence via middleware or any other mechanism was not tackled in this research and remains an area of future development. The relationships between the necessary development efforts are shown in figure 1.1.

A side effect of choosing a module of implementation at runtime is the notion of dynamic linking. Dynamic linking refers to delaying the linking of libraries to an executable from compile time to runtime [12].

The idea of "hot swapping" implementations through the use of dynamic linking has existed since the earliest day's of modern computing [12]. Since the 1960's, researchers have understood the advantages of the ability to dynamically load and execute sections of code at run time, unknown at compile time. A more modern related technology also required for runtime inclusion of new code is dependency injection. Dependency provides a mechanism to decouple the client program from a given implementation [9]. There are many motivations for these technologies. Applications may require updates, but cannot be shut down for safety or security reasons, such as critical systems or long-lived server applications [7]. Other applications require dynamic class loading as a normal functioning of their everyday operations, such as highly modular applications that support plug-ins [9]. Such an application's power and versatility lies in the ability to "hot swap" sections of executable code previously unknown to the program and unavailable at compile time.

1.1 Motivation

The increase of information flow will be the weapon of choice for tomorrow's warfighters in the battlespace of the future. One way to facilitate this increased communications ability is through hi-bandwidth, directional links, such as lasers or microwaves. However, these direction oriented links are susceptible to problems not present in the wireless domain such as link winking (connections between nodes coming in and out of service) and other physical limitations such as a node being limited to the number of other nodes with which it can communicate (finite number of lasers on each node, etc.) Previous research in the area of Net-Centric Warfare has sought to overcome these obstacles and increase of information flow through research efforts in strategic buffering, to overcome link winking; dynamic topology, to best utilize limited communications resources at each node; fault tolerance, and stochastic estimation control of queues.

Several of these research efforts were implemented and studied using OPNET Modeler, a network research and development environment. Unfortunately, these research efforts were carried out independently. Each researcher implemented their own protocol the best way they saw fit, with no regard for future integration or compatibility with other systems. The only grain of compatibility between the different protocol implementations was the adherence to the OPNET Modeler development paradigms of network models (scenarios), node models, and process models.

1.2 Problem Statement

The AFIT research community posses a rich and powerful tools to evaluate the effectiveness of any single network protocol through the use of OPNET Modeler; however, there does not exist a tool or technique for studying unrelated OPNET protocol implementations in conjunction with each other. The research community needs an OPNET toolkit that allows “hot swapping” functionality of OPNET Modeler implemented algorithms. This proposed new environment would facilitate “hot-swapping” between various networking protocols and re-combining them into different configurations, allowing for a careful study of the algorithms’ interactions. This proposed framework would be modeled after other plug and play environments such as Apache’s Tomcat [30], Autumn [1], and Spring [28]. These software packages all share the thread of commonality that users must provide implementations for critical pieces of infrastructure, but the infrastructure its self is not dependent on the implementations that the user provides. In fact, many of these frameworks allow for a simple “drag and drop” interface to specify new implementations, with limited other configuration required.

1.3 Research Objectives

The strategy this thesis describes, to provide the hot swapping ability of OPNET Modeler implemented networking algorithms, is dependency injection in conjunction with DLL’s, as described by Martin Fowler [9] [10]. By turning OPNET Modeler into a dependency injection environment, users may realize the full benefits of algorithm inter-operability and swappability.

The motivation for focusing on algorithms developed exclusively with OPNET is that it allows the toolkit to capitalize on the fundamental similarities of the implementations from a technical standpoint. Providing for integration of generic algorithms or algorithms developed with another platform in mind, such as NS-2 is, in fact, a different problem. A contrast of these seemingly similar, but vastly differing problems appears in chapter 3.

The purpose of this research is twofold:

- Develop an inversion of control container for OPNET utilizing dependency injection to facilitate implementation hot swapping.
- Exercise the framework to show its utility.

The framework will be exercised through the design and implementation of two applications. The first application provides an application programming interface (API) for the following:

- Building OPNET scenarios.
- Selecting networking algorithms, post compile time through the dependency injection framework.
- Running the simulation and collecting the data.

The second application is a unit testing framework that allows for the unit testing of OPNET process model implementations and provides for the following:

- Generation of different test cases.
- Automation of test case execution.
- Automation of axiom enforcement.

1.4 Implications

The programmatic control of OPNET simulations, coupled with the ability to remove the dependencies between OPNET and key protocol implementations, opens the door for several new uses for the OPNET Modeler development environment. First, simulations can be run as external libraries, called from other processes. The configuration of the simulation, as well as the results of the simulation can all be controlled programmatically through an API. This provides for new applications that require use of OPNET simulations as part of a larger research effort. Second, the dependency injection mechanism allows for the effective unit testing of process model implementations as described in chapter 4 of this thesis. The ability to effectively unit test OPNET implementations could facilitate a move to new simulation development methodologies, incorporating lessons learned from the software engineering community about software testing and agile development methodologies. These new methodologies could show a marked increase in protocol implementation quality, and allow for faster and more accurate validation of results.

1.5 Preview

Chapter 2 discusses other relevant work in software library design, network protocol research, and dependency injection testing frameworks. Chapter 3 describes the design methodology for the framework, with important design decisions and tradeoffs noted and explained. Chapter 4 describes the inclusion of two strategic buffering protocols into the toolkit. Chapter 5 discusses experiments designed to verify the toolkit's protocol implementations. Chapter 6 discusses implications of the framework as well as directions for future research.

II. Background

Many advances have been made in the area of modular object-oriented toolkit design. Put simply, a software toolkit is "a collection of classes that provides useful functionality but does not define the design of an application [15]." There are several critical design philosophies that toolkit developers grapple with to provide maximum functionality to the client program while reducing the amount of low level manipulation required. "The design of software toolkits embodies a fundamental tension. On the one hand, it aims to reduce programmer effort by providing prefabricated, reusable software modules encapsulating common application behaviors. On the other, it seeks to support a range of styles of application behavior [6]." This chapter is divided into three fundamental sections: section one describes the fundamental principals of software library design that influenced the libraries in this thesis and related design philosophies used in other application domains. The second section describes the philosophy of "Dependency Injection" used extensively in this thesis. Lastly, the third section provides a brief overview of the two previous theses integrated using the framework.

2.1 Toolkits and Library Design Principles

2.1.1 Polylithic Versus Monolithic Designs. Benderson [2] demonstrated in the area of graphical toolkit design that this fundamental tension is often manifested in a design decision between a polylithic and a monolithic design methodology. Polylithic designs are characterized by many small classes with limited—but clear—functionality. This design favored the client aggregating these many small classes at runtime into a larger, domain-specific module. Monolithic designs are characterized by a few large classes containing core and generic functionality. These designs favor the client application specializing the larger generic classes through inheritance. Benderson concludes that the optimum design methodology depends on the target audience of the toolkit. Polylithic designs offer the client "greater freedom to design, modify, and extend and maintain the toolkit [2]..." Benderson notes that this comes at the cost of the associated classes representing more abstract constructs and places a greater burden on the client to recognize the relationship between the numerous objects. On the other hand Benderson notes that monolithic designs are far better suited to novice users. Monolithic designs provide for large concrete classes that offer a broad host of functionality that inexperienced users can simply specialize through inheritance. However, Benderson notes that this severely limits the toolkit user from modifying critical policies set forth by the toolkit designers. Thus, he concludes that if the toolkit designer is in a position to accurately anticipate the future requirements of the toolkit, then a monolithic design is better suited. However, if optimal reusability by expert users is the goal, a polylithic design methodology is more appropriate [2].

2.1.2 The Click Modular Router. Kohler [18] describes a similar toolkit design philosophy with the design of Click, a modular router. The design motivation of Click centered around the need for a flexible, extendable, software router for network administrators and research groups to extend an existing router with new functionality. Kohler describes the Click architecture as a polylithic. "Click routers are built from fine-grained components; this supports fine-grained extensions throughout the forwarding path [18]." As Kohler explains, this has the side effect of each of the fine-grained components, called elements, to posses small and simple interfaces. However, because the router is built from such small building blocks, users can easily extend the functionality of just a small piece without dealing with the complexities of other elements. Kohler's polylithic router architecture facilitates UNIX style composition, with many small re-usable programs linked together in various combinations to form different high level applications. However, this style of architecture imposes a performance penalty. Kohler noted that handing of the packet between each of the Click routers' subsystems requires roughly 70 nanoseconds of overhead, with a total overhead of a millisecond for processing overhead of the entire router. Special tools were required to circumvent the overhead costs.

2.1.3 OSKit. Others have tackled toolkit design issues in operating systems development. Ford [8] describes another important aspect of toolkit design through the use of OSKit: implementation encapsulation through well-defined interfaces. OSKit provides a "bare-bones" framework for operating systems development, with the intent that the users of OSKit will then have more time to concentrate on the "real" issues of operating system development. As a part of this minimal framework, the designers have to provide generic implementations of common operating system procedures, such as drivers, file systems, and network protocol stacks. However, Ford notes that these bodies of code–borrowed from industry–are often rapidly changing. Rather than assimilate the volatile code into OSKit and require frequent revisions and rewriting of the changing subsystems each time a new release is issued, Ford introduces the concept of "glue code" that servers as an interface between the volatile subsystem and the framework of OSKit. This well defined interface provided a buffer that required a small amount of maintenance, at the benefit of re-using existing implementations without affecting the rest of OSKit's structure. Thus, with the "glue code," Ford was able to make liberal re-use of existing software subsystems, mostly or even completely unmodified from their original form, and provide this added functionality to clients of OSKit [8].

2.1.4 Oasis. Madhyastha introduces a toolkit, Oasis, for utilizing network overlays, such as Akamai, Kazaa, and Bittorrent. Madhyastha describes the motivation as the need for "a system and a toolkit that enables legacy operating systems to access overlay-based packet delivery services [20]." Madhyastha describes four key concepts that need to be addressed in all toolkits, and particularly network related toolkits:

1. Fine-grained control
2. Do no harm
3. Extensibility

4. Deployability

Interestingly, Madhyastha shows that the design goal of fine-grained control by the user does not necessitate large numbers of fine-grained objects as in the polylithic model. Instead Madhyastha favors a large, coarse grained architecture with more complex implementations. This is also motivated by the design objective of extensibility, to which the larger classed monolithic model lends itself. Additionally, Madhyastha raises the issue that the toolkit should not incur a significantly more overhead, either in execution time or memory requirements, than if the client had not utilized the toolkit. Madhyastha's research shows that their toolkit incurs minimal overhead when utilizing the additional layer for downloads. However, because of overhead in dealing with an extra layer of abstraction interfacing with the operating system, Madhyastha notes that it is unusable at upload speeds of 3Mbs. Although Oasis provided the benefit of fine-grained control and extensibility, it provided these benefits at the cost of massive overhead in certain situations [20].

2.1.5 Intelligent Network Configuration Optimization Toolkit. The Intelligent Network Configuration Optimization Toolkit (INCOT) was developed by Stottler Henke Associates for the Air Force [25]. The primary goal of INCOT "is to provide an intelligent interface to OPNET products, allowing rapid design and optimization of communications networks without requiring the user to have programming skills or knowledge of the underlying OPNET simulation products [25]." Richards and his associates provide a platform for Air Force network engineers to build optimized networks that meet specific requirements and achieve the goals and objectives set fourth by the engineers without requiring the users of the system to understand the underlying complexities of network simulation tools such as OPNET. Richards and his associates abstract the complexities of OPNET through the use of artificial intelligence (AI). The artificial intelligence algorithms use the input rules, objectives and policies from the user and construct and configure the optimized network simulation in OPNET. Through its interface with OPNET, the toolkit can then return the visualization and results of the simulation back to the user for review. Thus, INCOT facilitates rapid development of network design for engineers unfamiliar with OPNET products, and perhaps more importantly, concentrates the paradigm of thinking in network design to that of the underlying requirements and goals of the network, not the physical design.

2.1.6 Visualization Toolkit. Schroeder [26] describes the design goals and important object oriented design philosophies surrounding the creation of his Visualization Toolkit (vtk). Schroeder et. al. take their object-oriented (OO) design methodologies from researchers such as Booch and Rumbaugh and implement the toolkit in the OO language, C++. Schroeder et. al. describe their philosophy:

> "One important lesson we learned is that building large, monolithic systems is detrimental to software flexibility. As a result, we wanted to create a sharply focused object library that we could easily embed and distribute into our applications....The key here is that the pieces must be well defined with simple interfaces. In this way they can be readily assembled into larger systems [26]."

Through their highly cohesive object library, Schroeder et. al. followed the polylithic design philosophy described by Benderson [2]. Schroeder et.al outlined five key principles central to the success of their toolkit: Interpreted Language Interface, Standards Based Design, Portability, and Simplicity. The interpreted language interface utilized by Schroeder allowed for quick application development and debugging of high-level parts of the system. The standards based design approach directly enabled the portability of the toolkit. Previous iterations of the toolkit utilizing inhouse technologies proved to be difficult to maintain and convince researchers in other organizations to develop. As a result, Schroeder decided to switch to a standard's based approach, resolving all the aforementioned problems. Lastly, the principle of simplicity enabled wider use by users of the visualization toolkit and ease of maintenance.

One interesting OO design paradigm that Schroeder et. al. deviated from is encapsulating data structures and the methods that operate on them into one class. While this is certainly good OO programming practice, Schroeder noticed that this "would result in excessively large objects [26]." Thus, Schroeder opted to separate out these two main components into separate objects for greater simplicity and to provide greater comfort to users.

2.1.7 ET++. Andre Weinand et. al. [31] developed ET++, which "is based on MacApp and integrates a rich collection of user interface building blocks as well as basic data structures to form a homogeneous and extensible system [31]." ET++ allows for the rapid generation of user interfaces in the UNIX environment. Weinand et. al utilize many fundamental OO programming concepts present in all OO, modular and extensible toolkits. The heart of the ET++ was a layered approach that allowed for the abstraction of details maintained lower in the class hierarchy, particularly those related to hardware specific implementations. Thus, in the 3-tiered architecture, the highest level presents the interface that users of the toolkit utilize. Here, common graphical elements found in all toolkits are implemented. Services that these top level implementations require are implemented in the middle layer, known as the abstract system interface which defines "the minimal set of low-level functions necessary to implement ET++ [31]." Thus, this middle layer provides a virtual windowing environment to insulate the top level ET++ implementations from low level platform related windowing system.

Beyond the layered architecture, ET++ makes use of many standard OO design patterns. Among these are the strategy pattern. The strategy pattern enables multiple forms of similar algorithms to be incorporated seamlessly by presenting a common algorithm interface to the client. This decouples the implementation of the algorithm from it's use in the client. The decision of which algorithm to execute can then be deferred to run time, allowing greater flexibility. ET++ utilizes the strategy pattern to incorporate various line breaking algorithms in text boxes. By utilizing the strategy pattern, the client of the toolkit can decide which line breaking algorithm to use easily without any recompilation or re-factoring of code. Similarly, developers can add new line breaking algorithms easily without modifying code in the client because of the decoupling properties of the interface.

2.1.8 Executable Protocols and OPNET Simulation Environment.

Lew et. al [27] developed a simulation framework and associated API for interfacing OPNET simulations with various networking protocols implemented externally to OPNET. Lew explains that verifying current protocol implementations involves building the equivalent process model in OPNET Modeler. This process is both time consuming and error prone, as it is difficult to establish that the process model developed in fact represents the real-world implementation. By building specialized OPNET process models and creating special .NET wrapper API's, third party code can make the appropriate calls to the API and thus be unaware of any OPNET couplings. Thus, OPNET can then simulate protocol implementations meeting the API specification. Lew notes a few shortcomings of the framework, mainly that the API is protocol specific–not generic, so verification engineers must make frequent modifications to the interface as new protocols are developed. Additionally, the API does not contain any mechanism for passing statistical data between the simulation and any external interfaces, so the framework dumps all data into a flat text file [27].

2.1.9 Distributed Link 16 Simulation Demonstration.

Ryan Cooper [4] sought to simulate distributed link 16 implementations in a real-world scenario using OPNET co-simulation. Cooper's simulation scenario centered around an air-to-air "dog fighting" scenario with four F-15s pitted against four enemy F-15s. A separate co-simulation written in Java simulated the pertinent scenario activities (node mobility, when packets should be generated, and the effect of packet drop/delay). This information was then fed to a NETWARS co-simulation (NETWARS is a simplified subset of OPNET Modeler tailored for military applications). The NETWARS simulation then returned the status of the various packets involved in the simulation (arrived, dropped, latency, etc.). The co-simulation functioned by utilizing the standard HLA interface implemented with an RTI. The NETWARS "Commander" component controlled timing between the Java dog fighting scenario and the NETWARS co-simulation through the RTI [4].

2.1.10 Middleware Based Approaches. Dadarlat [5] describes the advantages of breaking down complex algorithms, such as routing protocols, related to networking toolkit design into modular classes that interact at the middleware layer. "The approach analyzes protocol functionality based on the idea of decomposing routing protocols into fundamental building blocks and identifying the role of each component." Dadarlat describes the current trend in protocol implementation is to produce "compact, monolithic" code "specific to each vender." Thus, to facilitate the design of new routing protocols, the design is decomposed into "fundamental building blocks," built as "collections of distributed objects." This new modular design allows new routing protocols to be assembled from the underlying find grained objects.

Critical to the success of Dadarlat and his associates' endeavors, was the creation of a binding model–determining how the most basic modules would be dependent upon, and thus coupled, to other modules. The success or failure of these basic modules depends heavily on the design and standardization of the interfaces between the modules. Dadarlat divides the modules in to three main categories: generic classes, routing component classes, and protocol specific classes. The generic classes provide services common to all protocols. The routing component classes were designed to be extensible and allow for the design of new routing mechanisms through the use of specialization. Lastly, the protocol specific classes sit at the highest level in the class hierarchy, and provide the remainder of the services necessary for the execution of the protocol that are unavailable at lower levels in the class hierarchy. Thus, by decomposing routing protocols in to smaller, well-defined modules, the designers of the toolkit were able to construct new routing protocols by simply re-combining the interactions between classes.

2.1.11 Sphere. Similarly to Dadarlat and associates, Stachtos et. al [29] also seek to reduce the time and complexity involved with the development of new routing protocols, by providing a developmental toolkit that abstracts away the fundamental routing protocol services into fine-grained re-usable modules [29]. To develop their toolkit, the designers began by studying the design of current routing protocols to ascertain commonalities between the various protocols. Similar to Dadarlat, common services were abstracted away into their own re-usable modules. Moreover, a binding model was developed, as with Dadarlat, to determine the nature of the interfaces between the modules. The modules were then organized into a reusable software development kit consisting of 3 types of modules: generic classes, routing component classes, and routing protocol classes. Similarly to Dadarlat, the generic classes sit at the lowest layer and implement services common to all upper layers. Such services include databases, thread-management, and network-connections. The routing component classes "represent the building blocks that can be used to construct different routing architectures [29]." One example is the routing database. The routing database extends the more generic database module from the layer below, and "provides information about network nodes and their associated paths, which can be used for performing route entry lookups [29]." Lastly, similarly to Dadarlat's design, the routing protocol classes sit at the highest layer in the architecture and implement services specific to each routing protocol. Thus, through their efforts, and the establishment of their binding model, the authors were able to construct numerous routing protocols with the development kit. Moreover, the authors were able to plug in different modules on the fly, and thus change the routing mechanisms dynamically.

2.2 Inversion of Control and Dependency Injection

Many large software frameworks make use of a design principle known as Inversion of Control (IoC) [10]. A framework implementing IoC is said to be an "IoC container." The hallmark of an IoC container is a module of the program looking up a particular implementation, a "plug-in", at runtime without maintaining a depen-

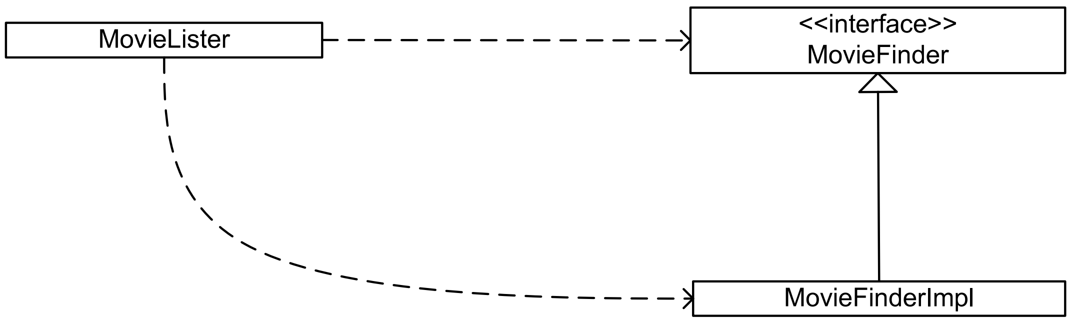

Figure 2.1: Fowler's "Naive" Example. In this example, MovieFinderImpl subclasses the MovieFinder interface. The MovieLister then depends on both the super and subclasses.

dency to the implementation. Fowler further defines the principles of IoC with the "dependency injection" pattern. Martin Fowler describes Dependency Injection (DI) as a method of "linking classes during configuration rather than compilation." [10] The motivation for the principle is the ability to specify differing implementations for behaviors based on the runtime environment–a fact unknown at compilation time. To demonstrate the effectiveness of the dependency injection architecture, Fowler [10] offers a concrete example in figure 2.1, with a paraphrase of his description reproduced here.

Suppose we had the application depicted in figure 2.1. The purpose of this simple application is to read a colon delimited text file of movies and their associated directors, and print out a list of movies by a particular director. The `MovieLister` class calls the `MovieFinder` interface. Then, `MovieFinderImpl`, the implementation of the interface, reads the colon delimited file and returns all movies and directors. The `MovieLister` filters through the information and returns only the ones with the specified director. In this setup, the `MovieLister` class depends on both the `MovieFinder` interface and, `MovieFinderImpl`, an implementation of `MovieFinder`. However, if the users want to expand the software to include a new type of `MovieFinder` (to read other file formats: XML, comma delimited, etc.), users must also modify `MovieLister` and recompile both classes. Fowler recommends reorganizing the classes such that a third

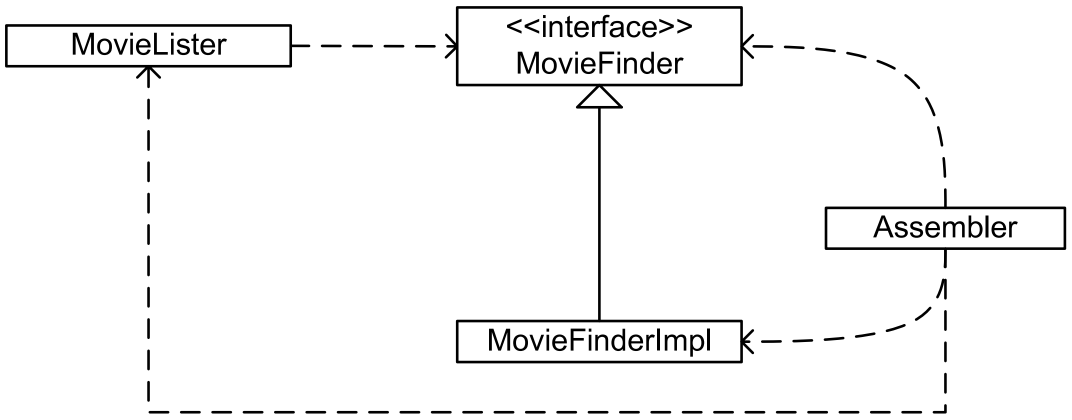

Figure 2.2: Dependency Injection Example. In Fowler's "Injector" example, the MovieLister depends solely on the MovieFinder interface. The assembler assumes the role of selecting a MovieFinderImpl and "injecting" the dependency into the MovieLister post-compile time. The introduction of the Assembler breaks the coupling between the MovieLister and the MovieFinderImpl, thus allowing new MovieFinderImpl's to be used without need for re-compilation of the MovieFinder.

party decides which implementation of `MovieFinder` the `MovieLister` uses (Figure 2.2).

With the inclusion of the `Assembler`, the only dependency the `MovieLister` retains is that of the `MovieFinder` interface. The assembler now decides, as a separate compilation unit, which implementation the `MovieLister` should use. There are several ways the Assembler can notify the `MovieLister` which particular implementation it should use.

1. Type I: Interface Injection. Define and use interfaces to perform the injection.
2. Type II: Setter Injection. Assembler sets a field in the client to the proper implementation.
3. Type III: Constructor Injection. When the client object is created, it requires the desired implementation as an initialization parameter.

No matter the form of injection used, the `Assembler` encapsulates decisions about which implementation the `MovieLister` should use. Thus, the `MovieLister`'s dependency on a particular `MovieFinderImpl` is said to have been "injected" by the `Assembler`. Now, users wishing to utilize the functionality of the `MovieLister` can install their own type of `MovieFinder` post compile time to correctly handle their custom file format (run-time environment); the `MovieLister` has no need for re-compilation.

The libraries described in this thesis utilize type II dependency injection (setter injection).

$\boxed{!}$ Dependency injection forms the backbone architecture of the libraries described in chapter 3 of this thesis. The libraries transform OPNET in to a setter injection based dependency injection framework, with OPNET acting as the MovieLister, and the toolkit, as a whole, functioning as the Assembler. The hot swappable implementations then fulfill the role of the MovieFinder-Impl's.

2.3 Related Thesis Work

The primary long-term goal of the toolkit is to incorporate network protocol related algorithms into one simulation. The algorithms that I will integrate are two different strategic buffering mechanisms.

2.3.1 Strategic Buffering. In the domain of directional wireless communication between nodes, the link between two nodes often "winks." That is there are not long periods of service, followed by long periods of no service, the link's connectivity continually changes as trucks, clouds, or other environmental conditions disrupt the link. Traditional TCP/IP protocols do not deal with the case of the "winking" link. In previous AFIT research, Maj. Duane Harmon showed that by buffering packets smartly, i.e. strategically, at the nodes on either side of a winking link, the efficiency of the network increases. New algorithms for handling the case of the winking link are still underdevelopment, and the toolkit libraries will allow future research to more easily compare and contrast the performance of different buffering mechanisms [16].

2.4 Conclusion

This chapter covered two main areas of interest in the background of this research. First, the chapter covered a general review of effective library designs in a variety of software systems in related domains. The main decision a library having to make is that of a polylithic versus a monolithic design. The chapter provided several examples of each with pro's and con's of each. Precedent indicates that monolithic designs contain relatively few numbers of large classes meant to be extended. Polylithic designs contain large numbers of small objects meant to be aggregated by the user to various purposes. Lastly, the chapter describes the basic principles of dependency injection as described by Martin Fowler. Dependency injection allows an application to delay the linking classes from compilation time to configuration time (runtime). Thus, the application can configure itself based on the runtime environment of an application.

III. Design Methodology

This chapter describes the fundamental design and implementation details of the toolkit libraries. The top level goal of the toolkit libraries, as described in chapter 1, is twofold:

1. Develop an inversion of control container for OPNET utilizing dependency injection.
2. Exercise the framework to show its utility.

The most important of these goals, to transform the OPNET Modeler into a dependency injection framework, allows the implementations of network objects in an OPNET simulation to be determined at runtime rather than compile time–a feature not available in OPNET. The other two requirements of the toolkit libraries, API's for building scenarios and controlling simulation execution are fundamentally already available from OPNET. Consequently, this chapter focuses on the second requirement and describes the design used to achieve this functionality with a discussion of necessary design tradeoffs.

Three factors the methodology aims to take into account are the following:

1. Preservation of OPNET's polylithic design.
2. Elimination of the coupling between simulations and key implementations.
3. Eliminate constraints caused by OPNET's proprietary nature.

The first section describes the OPNET scenario generation mechanism. The second section describes the design and important implementation details of the dependency injection mechanism, with a case study of a simple example. The third section describes the API design for controlling simulation execution, and the last section describes the mechanisms for statistic collection.

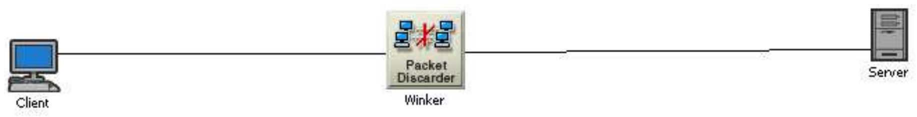

Figure 3.1: Basic OPNET scenario. This scenario contains a client and a server connected by an intermediate packet dropping mechanism.

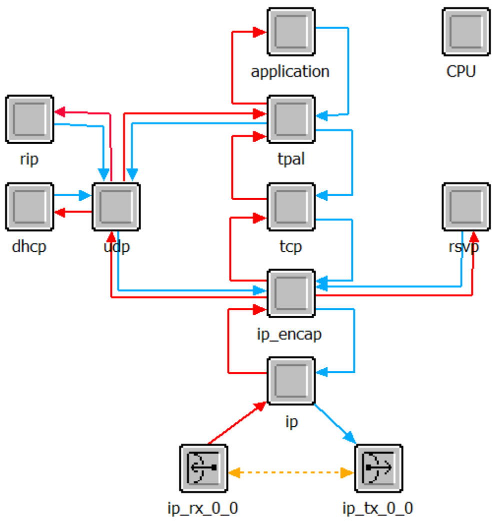

Figure 3.2: This is a basic OPNET node model. Similarly to the Click Modular Router, boxes represent separate modules of processing ability inside a particular node in a scenario.

3.1 OPNET Fundamentals

3.1.1 OPNET Organization/Operation. The OPNET simulation environment introduces three distinct levels of development that are pertinent to implementation decisions of the toolkit. The first and highest level of development is the "Scenario" level of simulation modeling. Figure 3.1 shows the scenario level of development, representing the fundamental network objects: clients, servers, routers, switches, etc.

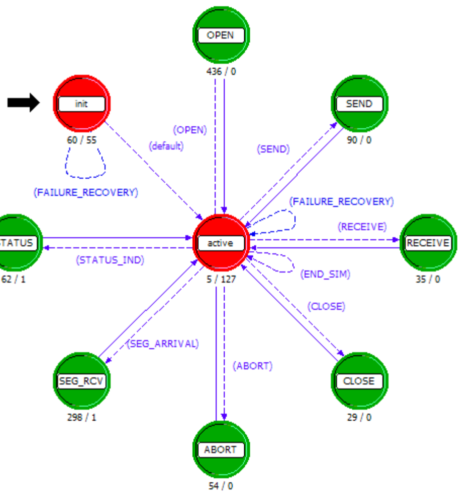

Figure 3.3: OPNET process model. Process models are state transition diagrams with executable proto-C, that executes when each state is reached.

Within each object at the scenario level, is a node model (Figure 3.2). Node models consist of one or more (typically dozens) of nodes. Similarly to the Click Modular Router, users wire together the nodes with a node model in various ways, yielding different behaviors of the corresponding object at the higher scenario level, be it a router, client, or some user defined network object. Lastly, the third and lowest fundamental paradigm of development is the process model (Figure 3.3).

An OPNET process model defines the implementation of each node in the node model. Process models are essentially user-defined state transition diagrams, with OPNET proto-C code contained in each state that OPNET executes when the path of execution reaches that state. Each state is a function that OPNET calls via a procedural style function call. After a state executes, the conditions of all of the state transitions to other states are then evaluated. If more than one path evaluates to true, then the simulation produces an error and terminates, otherwise the simulation execution follows the single path that evaluated to true and the OPNET kernel executes the next state's corresponding code.

3.1.2 Existing Protocol Implementations. Previous research efforts implemented the existing protocols that we wish to incorporate into the toolkit in this fashion, using one or more nodes within a node model defined by their respective process model implementations. Thus, to alter the implementation of a protocol is to change the implementation of one or more of the underlying process models of the nodes in a given scenario object's node model.

3.2 Specifying Process Model Implementations With Dependency Injection

3.2.1 Motivation. Ideally, developers performing protocol research want to change the implementation of a node (i.e. it's process model) post-compile time. In terms the OPNET domain, this research defines post-compile time as the following two-fold condition:

1. A node's process model has already been compiled, including all child processes.
2. The node model that contains the node in question has already been "saved" i.e. cannot be modified at runtime.

This first condition exists because one could conceivably define each of the process models under research as OPNET child processes. A root process could dispatch all network traffic to a specific child process depending on which model was being tested. While having the desired effect of "swapping" the node's implementation, this is still a solution for the specific case; OPNET requires that all child processes be "declared" for purposes of compilation. Adding child processes introduces a dependency between the OPNET simulation and both the protocol interface and the protocol implementation; as the OPNET documentation states:

> "A complete list of child process models that it [a parent process] intends to instantiate must be declared prior to simulation as part of the process model definition. Attempting to create a process based on a process model that is not declared may result in an error during simulation [23]."

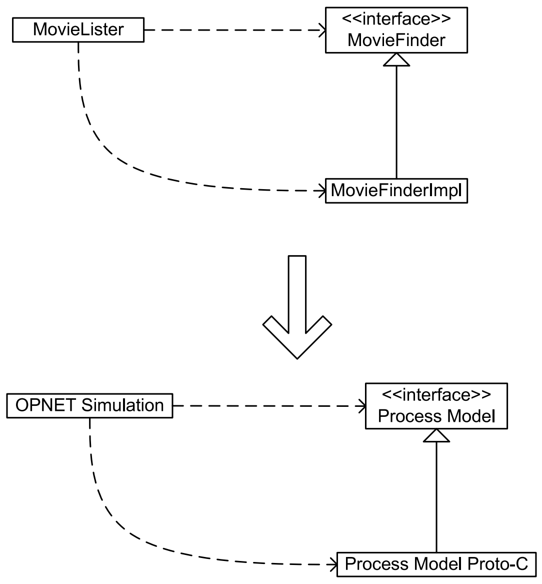

Figure 3.4: OPNET Dependency Model. This is the dependency structure that results from utilizing the mechanism of a root process delegating to one or more sub-processes as a strategy for dynamic algorithm selection. The dependency structure is identical to the structure of the "Naive" example referenced in Chapter 2.

This technique would require process model re-compilation, and thus this method will not work for post-compilation specification of process model implementations. Figure 3.4 highlights the dependency structure caused by a root process delegating to sub-processes to achieve the effect of run-time implementation selection.

Moreover, if the OPNET simulation were dependent on both the implementation interface, and the implementation itself, additional implementations would cause the simulation to likewise depend on each new implementation. The elimination of these dependencies is critical, and described in the next section.

The second condition exists because one could simply adjust the node specification to be defined by the target process model desired for testing. However, this is not a viable option because a distinctly programmatic interface for specifying the implementation to use is necessary for practical protocol examination. OPNET provides a programmatic interface for defining node models (External Model Access or EMA), however, rebuilding an entire node model from scratch solely for the purpose of changing a node's implementation is wasteful, akin to rebuilding a house for purposes of changing the light bulbs. Dependency injection solves all of these problems, by installing the equivalent of a standard socket, to which a user can install any standard light bulb (implementation) into the socket without re-building the house.

3.2.2 Alternate Methodology 1. Chapter 1 alluded to the problem of integrating generic process implementations, or implementations developed for another simulation platform, such as NS-2. This problem is substantially different from the problem tackled here. In fact, Lew et. al. tackled this problem specifically in [27]. Their solution involves creating a wrapper for certain OPNET kernel procedures with a specific custom API for each implementation desired for integration with OPNET. This research is different because it involves taking an external algorithm and integrating it with OPNET without creating coupling between the algorithm and OPNET. The problem solved in this thesis with dependency injection involves taking an implementation developed with OPNET and decoupling it from the simulation without significant re-factoring or loss functionality. Lew's solution is case-specific, with frequent modifications of the interface API necessary to maintain compatibility. The methodology discussed in this thesis depends solely on OPNET's process model methodology, which is stable from version to version.

3.2.3 Dependency Injection. The principles of dependency injection (DI) and the more general term, inversion of control (IoC), have long been used in many frameworks for the ability to defer establishing dependencies from compile time to a configuration period at run time. Moreover, these principles reduce dependencies of components on specific implementations of other components in a given system [10] [21]. Frameworks such as Autumn (C++), Spring (Java, .NET), and PicoContainer (Java, .NET) all utilize DI and IoC [1] [22] [24] [28].

Thus, in order to remove the coupling between the OPNET simulation and the process model implementation, a module external to the simulation must assume control of choosing the appropriate implementation (in this case, the toolkit libraries) and provide the implementation to the OPNET simulation. This achieves the end goal

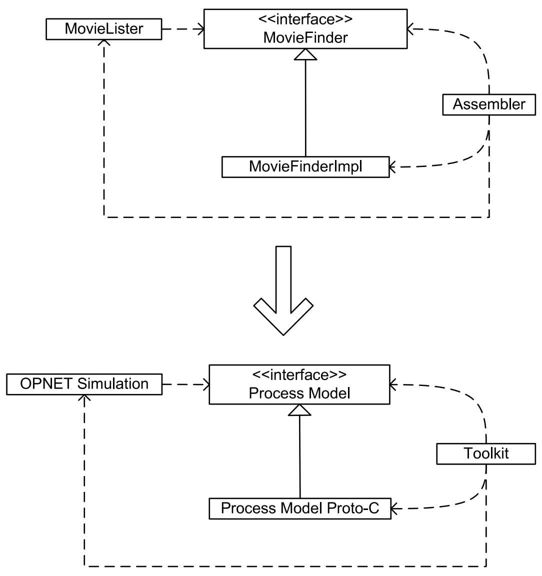

Figure 3.5: OPNET Decoupled Dependency Model. This is the dependency structure that results from the use of the toolkit with dependency injection for dynamic algorithm selection. The OPNET simulation is no longer coupled to the algorithm implementation.

of the OPNET simulation being dependent only on an interface for the implementation, and having no compile-time dependency on the implementation. The toolkit utilizes type 2 IoC, called "setter injection" by Martin Fowler [10]. In this method of dependency injection, the assembler module (a toolkit component) calls a setter method on the client (the OPNET simulation) and sets a field with a reference to the appropriate implementation. In the OPNET simulation domain, use of an OPNET Co-Simulation with an Esys (External System) interface achieves effect of setter injection. The framework controls the execution of the simulation as an external library, and the Esys interface allows for information passing (through shared memory) between the simulation and any external module utilizing the Esys API (Figure 3.5).

3.3 Process Model Refactoring

In order to utilize the advantages of dependency injection in the context of the OPNET simulation platform, the implementation that is to be injected must be available to the assembler for selection and subsequent injection through the setter in-

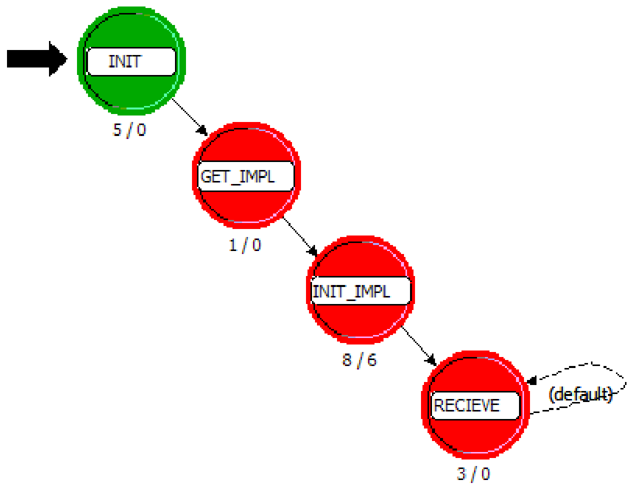

Figure 3.6: The OPNET "Hook" Process Model. This is the process model that the toolkit users must install in to processes that are candidates for integration with the toolkit. This "dummy" process delegates all incoming interrupts received by the process to the externalized process model via the ProcessInterface.

terface. Because the testing environment is external to OPNET, the implementation must be externalized as well. In the OPNET domain, the implementation is a process model, as described earlier. The process model shown in figure 3.6 replaces the original process model in the node under test. This new model stalls in the "GET_IMPL" unforced state until it receives an interrupt from the Esys interface delivering a reference to the implementation that is to be used for this particular execution. After this has occurred, interrupts received by the node under test trigger a call to the externalized implementation.

All external code implementations for process models conform to the simple interface shown in listing 3.1. The interface's most important method is the `process Interrupt()` method which takes no parameters and returns void. The framework guarantees that this method will be called by the simulation every time the node under test receives an interrupt of any type. The interrupt type and other associated information need not be specified in the interface because the OPNET kernel maintains this information, and the external implementation can simply ask the kernel

```
class State;
class ProcessInterface
{
public:
    virtual void processInterrupt()=0;
    virtual void initialize()=0;

    ProcessInterface();
    virtual ~ProcessInterface();
};
```

Listing 3.1: Process Implementation Interface. This is the interface to which all process implementations must comply. The `initialize()` method is guaranteed to be the first method invoked by the toolkit. Anytime the simulation invokes an interrupt to an object in the simulation, the simulation calls the `processInterrupt()` method.

for any associated information it may need for proper execution. Additionally, the interface contains an `initialize()` method which is guaranteed to be called by the framework before the first call to `processInterrupt()`. This allows the underlying implementation to perform any simulation related initialization procedures, such as initializing state variables based on externally set model attributes.

3.3.1 The framework State Pattern. Strictly speaking, all that is required of an implementation to be used by the framework is that it complies with the afore mentioned interface. The implementation details of the externalized OPNET process model need not bare any resemblance to the original OPNET implementation. However, under the principle of least representational gap, developers will often find that the best refactoring of the original OPNET process model is to the "State Pattern" as described in [15].

The toolkit contains several supporting modules that make this transition seamless. First, the software contains a `StateManager` that implements the `ProcessInterface` interface. The `StateManager` provides acceptable default implementations, with the `initialize()` method likely being the only method with cause for users to override. Second, the framework contains a `State` class that handles logic

Table 3.1: Refactoring OPNET Modules. This table shows the relationships between standard OPNET modules and the corresponding modules in the framework.

Refactoring OPNET Modules

OPNET Module:	Refactored Module:
States	Subclass `State`
State Variables	Struct or `StateVariable` class
State Transitions	Handled by each state
Function Block	`Utilities` class
Process Model	Subclass `StateManager`

associated with forced and unforced process model states. Developers need only to subclass `State` and provide implementations for the enter and exit executives. Lastly, because the implementation handles the state transitions and not the OPNET kernel, the states themselves must inform the `StateManager` of the next state to execute, by simply calling its `setCurrentState()` method. Table 3.1 summarizes the necessary refactorings.

Figure 3.7 shows the UML for the refactoring of the original OPNET process model containing only an "INIT" state and a "Wait" state.

3.3.2 Stochastic Packet Dropper: A simple proof of concept. In order to verify the methodology, the concepts discussed previously are demonstrated in the context of a simple node model containing one single node: a packet dropping mechanism. Thus, the required refactorings related to the packet dropper are identical to those required by any other node model and thus serves as an archetypal blue print for how to proceed in future protocol refactorings. The initial process model (Figure 3.8) contains four states: INIT, DROP, WAIT, and IDLE. In order to refactor the process model, I re-create the states and the state transitions contained in the process model utilizing the state pattern (Figure 3.7).

Thus, once all of the code that was once in the OPNET process model has been converted to an external state pattern based module, I replace the old OPNET process

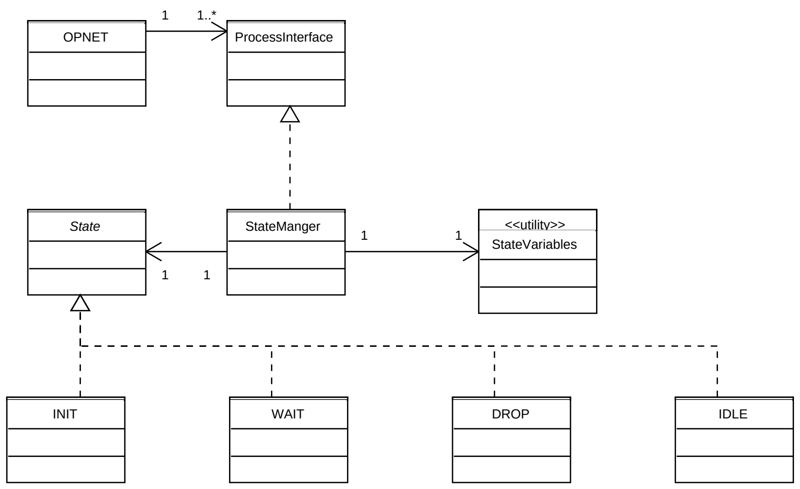

Figure 3.7: OPNET Process Model Externalization. This UML represents the necessary refactoring of process implementations (models) for use with the toolkit. The refactored implementation assumes control of state transitions via the State pattern as described in [15].

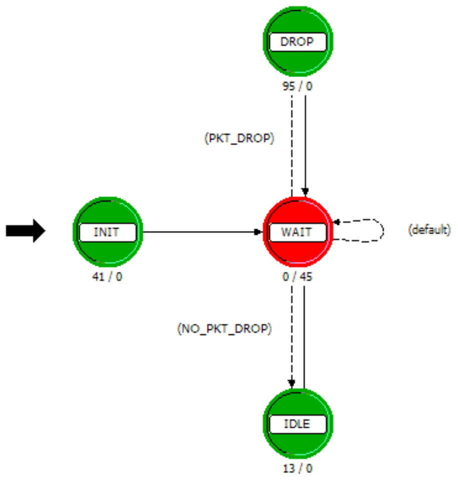

Figure 3.8: “Packet Dropper” Process Model. This is the process model by Maj. Harmon that implements a packet dropping mechanism based on various random inputs.

model with a simple “hook” process model that retrieves the interrupts generated by simulation activity from the OPNET kernel and passes it to the interface (Figure 3.6).

The code in the “hook” simply invokes the “processInterrupt” method of the interface:

```
manager->processInterrupt();
```

The specific mechanism for providing the implementation to the hook process model falls on the shoulders of the application layer, and is discussed in Chapter 4.

3.3.3 Methodological Criticisms. There are two ways to view the methodology this chapter described. At best, the methodology is a clever use of the features available in OPNET, taking full advantage of the most powerful options afforded by OPNET, mainly the ability to execute arbitrary C/C++. However, critics could argue that the methodology is an exploitation and gross abuse of the features available in OPNET. To properly implement the functionality this methodology affords without “exploiting” OPNET features would require modifying the mechanism by which the OPNET interacts with the process implementations. Because of the proprietary nature of OPNET, these modifications are not easily possible.

However, there are other motivations for utilizing the “out-of-the-box” features to implement the methodology. The major advantage is that the features utilized the methodology’s implementation are all standard, public, published, and well documented interfaces. Even if the interfaces are not ideal for our uses, the utilization of standard, published, and publicly facing interfaces allows the methodology to make no assumptions about internal implementation details of the OPNET kernel. Moreover, OPNET kernel developers have a motivation to keep previously published interfaces stable to allow backwards compatibility and facilitate a stable development environment for users. There is no such motivation for the unpublished internal interfaces. Thus, there is a higher probability that this methodology will work with future versions of OPNET than if internal interfaces had been utilized. Lastly, because this methodology does not interfere with OPNET's internal mechanisms for executing process implementations, the burden for ensuring the large body of legacy process implementations remain compatible fall on OPNET, not the methodology's implementation.

3.4 Conclusion

Through the externalization of process models into stand alone C/C++ and through customized "hook" process model implementations, a framework can realize the full benefits of dependency injection. The incorporation of dependency injection into the OPNET Modeler development environment decouples the OPNET simulation from the process implementations that define the behavior of network objects. This enables an external entity to select a particular process implementation for any given simulation execution post compile-time.

IV. Two Example Applications

In order to best demonstrate the utility of the mechanisms described in the previous chapter, I demonstrate the design and implementation of the OPNET Network Protocol Testbed (OP-NPT), a collection of libraries that enables programmatic specification of OPNET scenarios, run-time selection of protocols, and execution of simulations. Building on these libraries, I then discuss OPNET-Unit, a library utilizing many of the toolkit libraries that facilitates OPNET process implementation unit testing. Accordingly, this chapter is divided into two sections. First appears the design and implementation of OP-NPT followed by the design and implementation of OPNET-Unit. Each section first presents a minimal set of top-level use cases for the library or application. The design and implementation of the portions of the library or application fulfilling each use case then follow.

4.1 OPNET Network Protocol Testbed

4.1.1 Overall Architecture. The OPNET Network Protocol Testbed (OP-NPT) under development must integrate with several other concurrent development efforts. The ultimate goal is for a single network simulation scenario to be run into two toolkits: one interfacing with OPNET, and another interfacing with Network Simulator 2 (NS2). The toolkits will provide the necessary services specific to each simulation platform for the running and management of simulations. Additionally and most importantly, the toolkits will provide a programmatic interface for all levels of net-centric warfare algorithm simulation integration: 1) scenario generation, 2) inclusion of specific implementations of NCW algorithms under research, and 3) management of simulation results. Both toolkits will yield results to a common interface, usable by a visualization framework (Figure 4.1).

4.1.2 OP-NPT Use Cases. The toolkit, as with all toolkits, is simply a collection of libraries–not a program proper with an associated "main" method. Craig Larman states that "use cases are text stories, widely used to discover and record requirements [19]." Thus, the use cases below are provided to gain a better

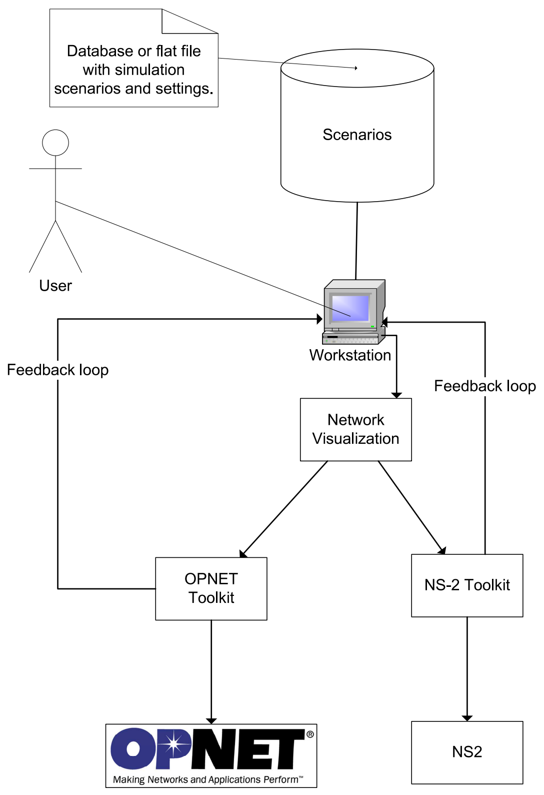

Figure 4.1: Overall Architecture. This diagram shows the relationship between the OP-NPT toolkit written using the OPNET IoC container libraries and other concurrent research efforts. The network visualization passes information to the toolkits, and the toolkits, intern, interact with their respective simulation platforms. Additionally, the ultimate goal is for a feedback loop to exist between the toolkits and the visualization, offering information about the simulation under execution.

UC1: Generate Scenario
Main Success Scenario: Developer begins development of their application with the proper toolkit libraries installed in their development environment. The developer make the appropriate method calls to the library's API. The program using the library produces the corresponding OPNET compatible scenario file.

Alternate Scenarios: The user uses the library incorrectly. When the library detects a problem, exceptions are thrown. The user's code handles thrown exceptions, or uncaught exceptions terminate execution.

understanding of the functionality that a user of the libraries should expect. Because OP-NPT is a toolkit, the use cases are not from the perspective of the user of polished application, but those of an application developer who wants to utilize the functionality of the OP-NPT toolkit libraries. The "main success scenario" is the best case usage scenario for interaction with the libraries, while the "alternate scenarios" describe exceptional cases.

4.1.3 Use Case: Automating Scenario Generation. OPNET already provides an external API for programmatically generating OPNET readable scenarios. The API, known as External Model Access (EMA) is essentially a large array consisting of all of the objects contained in the simulations and objects and sub-objects representing all the related configuration information associated with various simulation components. Unfortunately, this style is not efficient or particularly human–readable. Even purely pedagogical scenarios consisting of a client and a server with a duplex link connecting them can contain 55,830 lines (Figure 4.2).

Thus, it is unreasonable to assume that clients of the toolkit will be able to make the appropriate calls to the complicated OPNET EMA API. To solve the problem of scenario generation, the builder pattern as described in Design Patterns [15] was employed. The builder pattern, as employed here, functions by providing a simple interface for constructing scenarios, consisting of a simple, coarse-grained API with such methods as `addLink()`, and `addNode()`. Concrete subclasses implement the specified

UC2: Specify Desired Protocols
Main Success Scenario: Developer continues development of their application with the proper toolkit libraries installed in the development environment. The developer utilizes the supporting toolkit libraries, such the `State` class and the DLL template, and produces the necessary toolkit DLL implementation of algorithms they wish to utilize. The developer then makes necessary modifications to config files, etc. User writes C++ program utilizing OP-NPT libraries to initialize the simulation, and specifies the desired algorithms with the necessary method calls.

Alternate Scenario1: The user provides a mal-formed DLL implementation. When the library detects a problem, exceptions are thrown. The user's code handles thrown exceptions, or uncaught exceptions terminate execution.

Alternate Scenario2: The user improperly modifies config file. When the library detects a problem, exceptions are thrown. The user's code handles thrown exceptions, or uncaught exceptions terminate execution.

Alternate Scenario3: The user specifies non-existent protocols. When the library detects a problem, exceptions are thrown. The user's code handles thrown exceptions, or uncaught exceptions terminate execution.

UC3: Running Properly Configured Simulation
Main Success Scenario: Developer continues development of their application with the proper toolkit libraries installed in the development environment. Developer make the appropriate calls to the simulation libraries to attributes of the simulation to execute, such as duration, promoted attributes from implementations, etc. The developer utilizes the API for executing the simulation and the simulation runs properly.

Alternate Scenario1: The user provides mal-formed initialization information. When the library detects that the simulation has terminated with an error code, exceptions are thrown. The user's code handles thrown exceptions, or uncaught exceptions terminate execution.

Alternate Scenario2: The simulation terminates with any error code. When the library detects that the simulation has terminated with an error code, exceptions are thrown. The user's code handles thrown exceptions, or uncaught exceptions terminate execution.

UC4: Collecting Results
Main Success Scenario: Developer continues development of their application with the proper toolkit libraries installed in the development environment. Developer runs the simulation according to use case 3. The simulation records all raw data. Via SQL queries, the user can then programmatically calculate and recover statistics.

Alternate Scenario1: The user provides mal-formed SQL queries. The libraries simply return the result of the queries to the user, be it the expected result set or an error message.

UC5: Interfacing with OP-NPT from Java
Main Success Scenario: Developer begins development in a Java environment. The developer utilizes the JNI enabled "Java API" wrapper library supplied with the toolkit. The user then configures, builds, and runs the simulation and collects results.

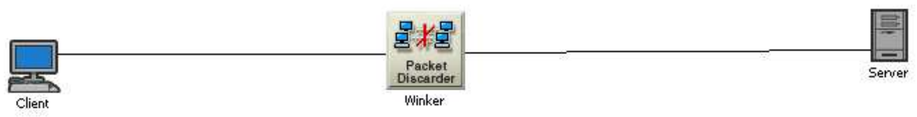

Figure 4.2: Simple OPNET Scenario. Even this small example requires upwards of 56 thousand lines of EMA code representation

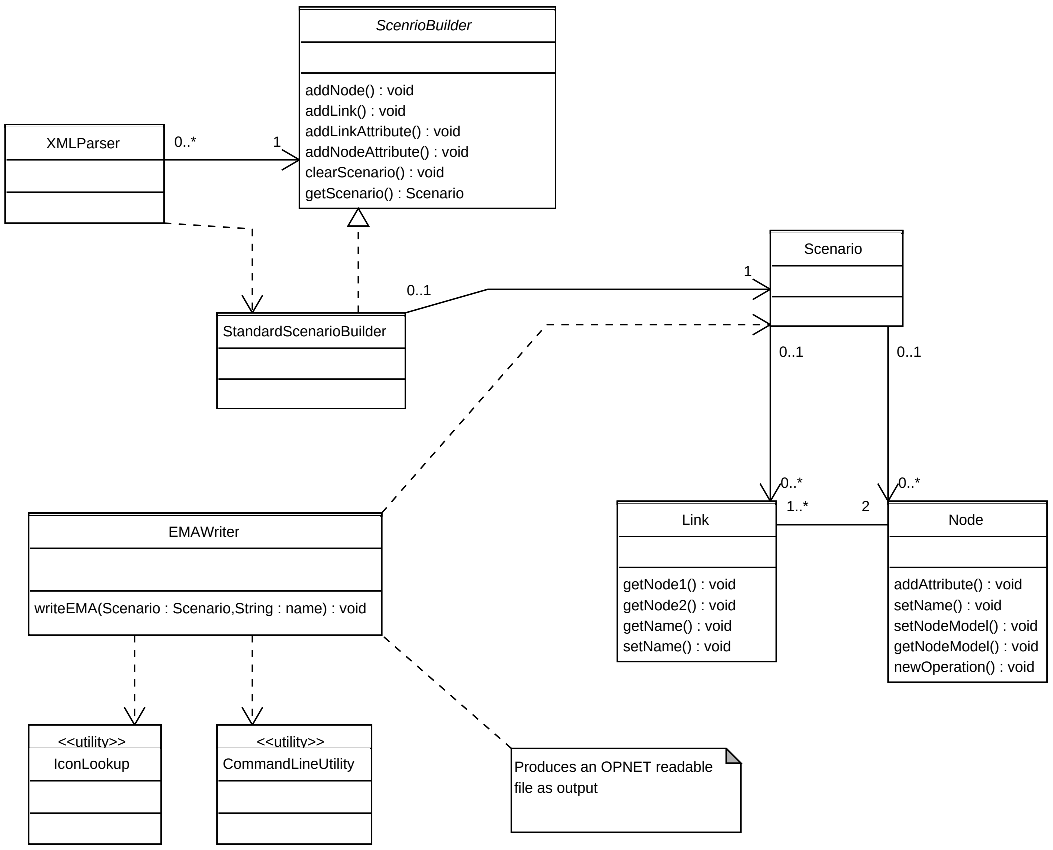

Figure 4.3: Scenario Generator UML. This diagram describes the collaboration of objects that generate OPNET readable network model files required by the OPNET kernel. The generator constructs an object representation of the scenario and then generates the appropriate EMA API calls via the Builder pattern as described in [15]

interface, producing a single Scenario object as a product. The Scenario object is then passed to a ScenarioBuilder utility that generates the appropriate OPNET EMA calls to produce the OPNET readable scenario file (Figure 4.3).

This approach has several benefits. First, because an intermediate Scenario object is created with the builder, various constraints can be checked and enforced during the build process, such as enforcing that a link cannot involve a nonexistent node. These constraints can be checked, with the appropriate object-oriented exception throwing mechanisms present in C++ utilized, giving the client code a reasonable opportunity to handle exceptional situations–a feature not present in the OPNET API. Secondly, by enabling an interface with a concrete subclass providing

```
ScenarioBuilder *myBuilder = new StandardScenarioBuilder();
try{
myBuilder->addNode(2,2,"node1","","txX","rxX");
myBuilder->addNode(2,2,"node2","","txX","rxX");
myBuilder->addLink("link1","ppp_adv","node1","node2");
myBuilder->addLinkAttribute("link1","attribute1","5000");
}
catch(addNodeException e){
//error handling code
}
catch(addLinkException e){
//error handling code
}
catch(addLinkAttributeException e){
//error handling code
}
```

Listing 4.1: Programmatically Building a Simple Scenario. This listing utilizes the toolkit API's and supporting libraries to build the scenario shown in Figure 5.1.

the implementation, subsequent implementations complying to the interface standard can be plugged in, enabling generation of scenarios with different constraint checks and implementations.

Listing 4.1 utilizes OP-NPT to build the scenario depicted in Figure 5.1:

Instead of the some 55,000 lines of code needed to generate the OPNET readable scenario, an application utilizing OP-NPT can achieve the same with just 10 lines of code. A decrease the number of lines of code, and a substantial decrease in complexity. Additionally, the objects in the toolkit libraries utilize the exception handling capabilities of C++, enabling the application to take appropriate action should a given action cause problems. For example, from lines 3-6 in Listing 4.1, several exceptions may be thrown. The author of the application can then choose to catch the exceptions (lines 8-16) and perform any pertinent actions he see's fit.

4.1.4 Use Case: Specify Desired Protocols. As stated previously, in the context of OPNET, the fundamental unit of a protocol implementation is the OPNET process model. Thus, to swap a protocol in a network object from protocol A to protocol B requires "swapping" one or more process models from one simulation execution to another. As previously described, the underlying dependency injection framework handles the swapping mechanism. However, in the context of realistic simulation scenario, two additional services are needed by the application layer. First, because the swapping of protocols will likely consist of multiple process models (or even multiple versions of different process models), a configuration management system is needed to aid the user of the application layer in the proper selection of process model implementations that correspond to the appropriate protocol implementations. Second, in keeping with the spirit of dependency injection, it is highly desirable that the addition of new process model implementations, and thus new protocols (or new versions of protocols) *does not require re-compilation of any code*, specifically the code involved in the configuration management.

4.1.4.1 Implementation Installation. When building simulations using OP-NPT, users must use specialized node models for the representation of node objects. These node models contain an OPNET Esys (External System) interface process in place of any process in the node model that the user wishes to be swappable. In the interface specification for the Esys process, the user specifies a name for the interface. When OP-NPT initializes the a simulation, it inspects the nodes and recovers all of the interface names and logs them in a table as required implementations. If the user does not specify an implementation for each interface, then the simulation cannot execute. Figure 4.4 shows the original node model for buffering strategy 1's proxy. The process of the node model that control the buffering mechanism are the port processes, and the "Proxy_Mempool_Manager." These five nodes combined together form the essence of the buffering protocol. Thus, to swap from a regular, non-buffering router to the buffering mechanism requires changing the original process implementations in each of the port process, and then adding a new node, the "Proxy_Mempool_Manager."

Because the toolkit implementation does not change the node model or build it on the fly, the designer of the node model must take care to ensure that the node

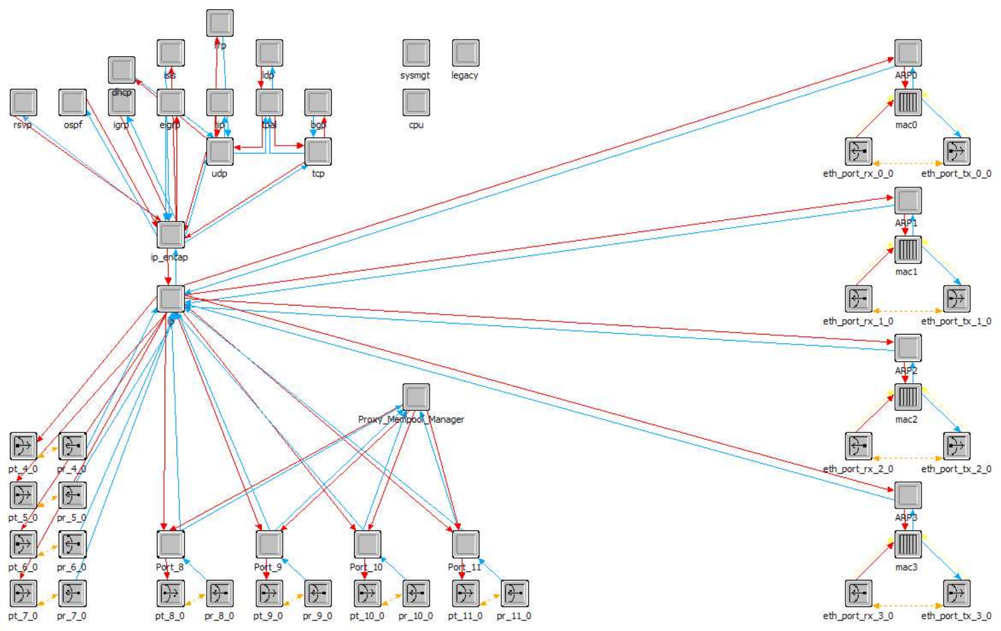

Figure 4.4: Buffering Strategy 1's Proxy Node Model. This is the original node model constructed to provide a buffering mechanism inside the router.

model design is sufficiently modular and re-usable. Fundamentally, the toolkit can swap any process model implementation A with any other process model implementation B, however, the interaction between the process models is dictated by the node model, not by the toolkit, and thus must be designed accordingly. The node models available from OPNET already follow this modular paradigm, with each layer of the OSI network stack having its own node representation. Moreover, different routing mechanisms are often encapsulated in their own processes within a router's node model. For example, a router that supports various routing methods often has separate processes for RSVP, OSPF, IGRP, and RIP.

Thus, to make the protocol implementation swappable, we change the pertinent processes with OPNET Esys interface processes. Each of the Esys processes contains a specialized "hook" process model that replaces the original process model. This specialized process model, in conjunction with the Esys Node it defines, advertises the type of interface it expects. The process model ensures that the externalized

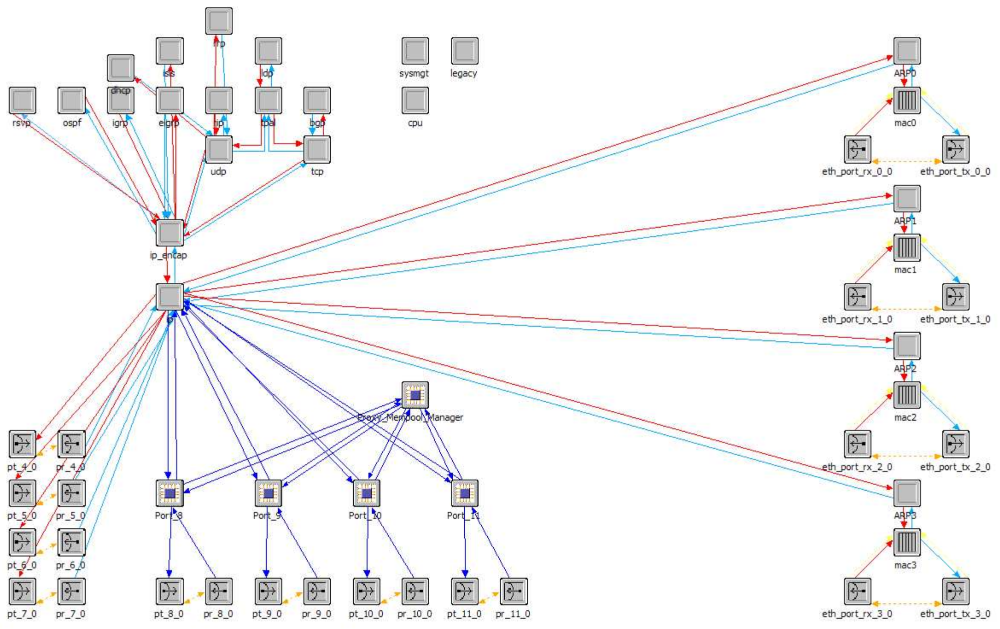

Figure 4.5: Modified Toolkit Proxy. This node model shows the necessary modifications on the original proxy for integration with the toolkit. The main changes required include replacing "swappable" processes with Esys interface nodes, defined by the toolkit's "Hook" process implementation.

process model is initialized when it is installed. Additionally, the process model serves as a final line of defense against mal-formed simulations (simulations that are run without all of the implementations specified), by preventing the de-referencing of NULL pointers.

4.1.4.2 Integrating Buffering Strategy 1. The first proxy buffering mechanism was implemented by Maj. Duane Harmon in previous AFIT research efforts. Figure 4.4 shows the proxy's node model. The following sections describe the incorporation of his implementations, using the methods described in Chapter 3, into OP-NPT. Harmon's buffering protocol consisted of just two process models: a memory pool manager and proxy port at each of the ports of the proxy enabled router. We modify the node model of the proxy router to include OPNET Esys processes in place of each of the pertinent aforementioned processes in the node model. The process models defining the process we wish to make swappable are replaced with the

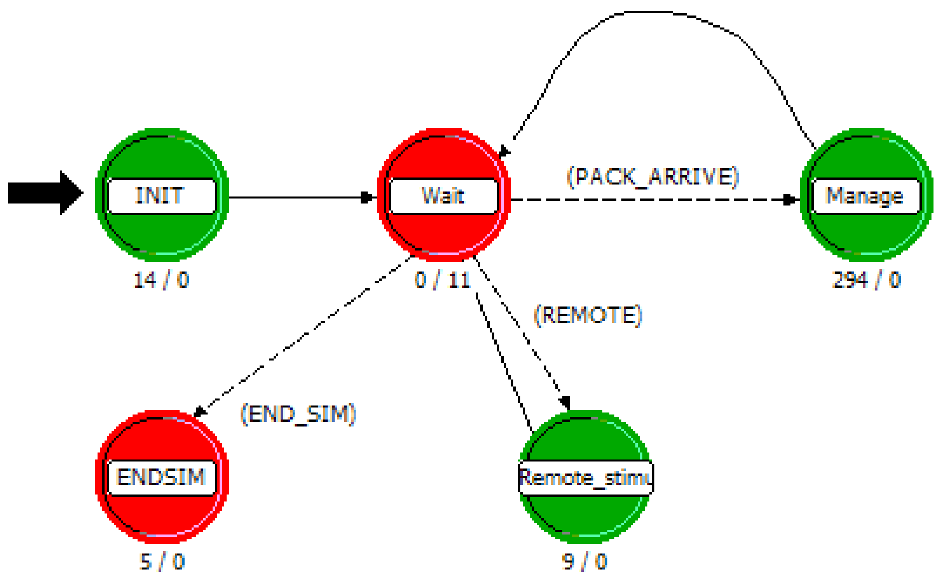

Figure 4.6: Buffering Strategy 1's Proxy Memory Pool Manager. This process model represents the original state transition diagram that defines the memory pool manager's process implementation.

specialized toolkit Esys process models discussed in Chapter 3. The two processes, the memory pool manager and the proxy port, must be externalized and re-compiled as DLL's as discussed in chapter 3 (Figure 4.5).

4.1.4.3 Proxy Mempool Manager Externalization. In accordance with the methods outlined in chapter 3, the process model of the "Proxy_Mempool _Manager" (Figure 4.6) must be externalized. Utilizing the OP-NPT state pattern templates, the process model was converted to a stand-alone object oriented C++ library. This library is then compiled into a DLL to allow dynamic loading by the operating system in to OP-NPT.

Figure 4.7 shows the resulting refactoring of the OPNET process model. Each state in the OPNET state transition model becomes a separate class of type "`State`" in the resulting refactoring. The process model's function block becomes a static utility, and the state variables likewise become their own class. The `StateManager`

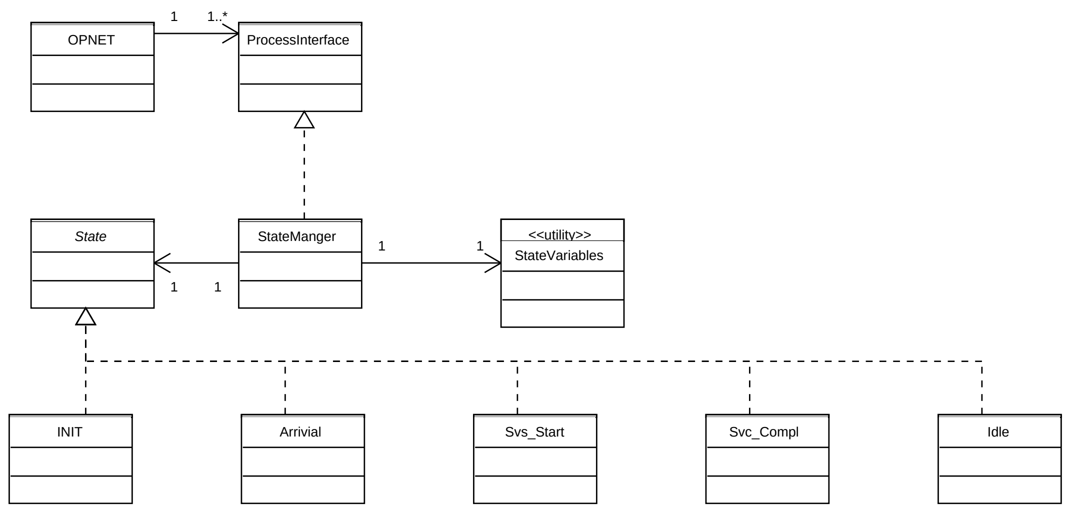

Figure 4.7: Memory Pool Manger Refactoring UML. This UML diagram shows the necessary refactorings of the original process model for integration with the toolkit. The implementations stay the same from process model to refactored externalization-only the overriding structure changes.

class manages which state the current externalized process model occupies, as well as managing access to the state variables and providing the implementation of the ProcessInterface interface to which the OPNET simulation interacts.

4.1.4.4 Proxy Port. Similarly to the Proxy_Mempool_Manager, we refactor the proxy port process model (Figure 4.8). As in the previous process model, all the states in the original process model, both forced and unforced, become separate classes in the refactoring (Figure 4.9). Usually in the code refactorings, the OPNET proto-C implementations from the process model states appear unchanged in the classes of the externalized refactorings. However, this was not the case of the proxy port process. Harmon made use of a supporting library that ships with OPNET as part of his implementation. This supporting library requires a call to an initialization function before any other function in the library can be called. In the context of the OPNET development environment, users are not required to call this method explicitly–the OPNET kernel takes care of this for the developer. However,

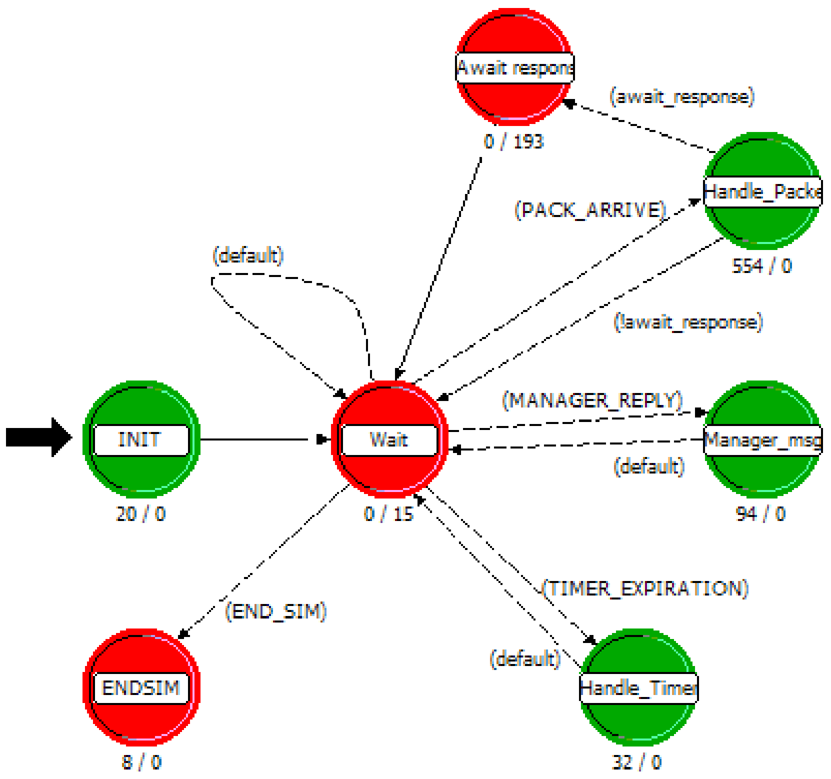

Figure 4.8: Buffering Strategy 1's Proxy Port. This process model represents the original state transition diagram that defines the proxy's port process implementation.

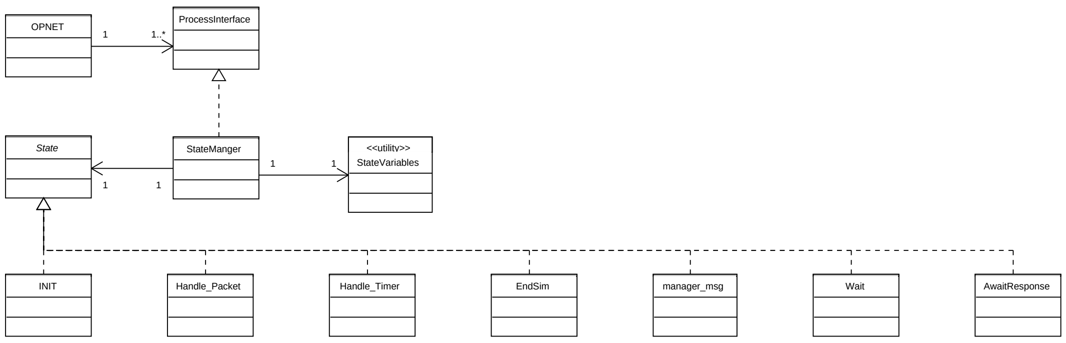

Figure 4.9: Proxy Port Refactoring UML. This UML diagram shows the necessary refactorings of the original process model for integration with the toolkit. The implementations stay the same from process model to refactored externalization, only the overriding structure changes.

in the externalized implementation, we lose this benefit, and must call any required initialization functions on OPNET provided supporting libraries explicitly.

4.1.4.5 Protocol Configuration Management. Constructs within OP-NPT allow the user to register new protocols, and specify the various implementations that define the protocol. OP-NPT can then perform configuration management services and ensure that the user can simply choose a given protocol and rely on the toolkit to install the right set of implementations into the underlying node model. Moreover, the user can specify multiple protocols, and the proper set of implementations will be installed. If two protocols are incompatible (requiring different implementations for a given interface) then the toolkit can detect this and notify the user.

4.1.4.6 Integrating Buffering Strategy 2. The second buffering strategy was implemented by Matthew Weeks, an intern, during the summer of 2007. Weeks was hired to investigate improvements to Harmon's strategic buffering mechanism. Weeks' improvements involved the quality of the implementation, as well as modifications to the protocol itself.

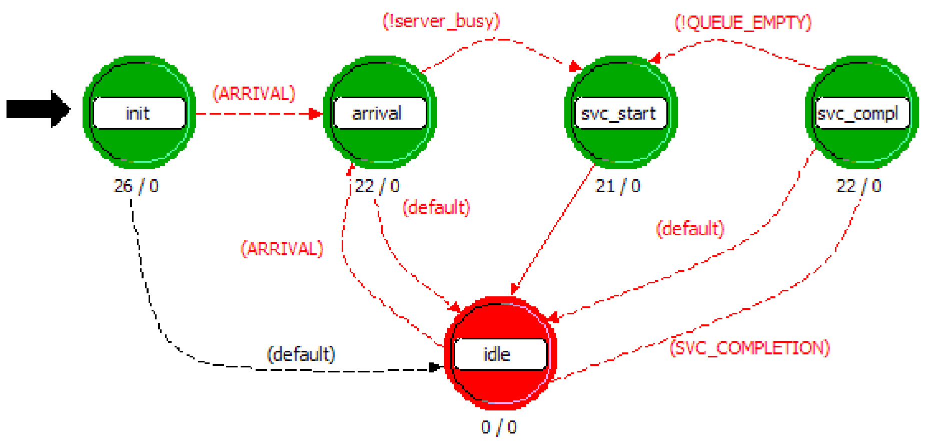

Figure 4.10: Proxy Queue Process Model. This is the proxy's Queue process model added in buffering strategy 2. The state transitions of the process model are unchanged from those provided by OPNET. However, several custom modifications were performed to the internal implementation of the queue.

4.1.4.7 Proxy Port Version Two. Weeks maintained Harmon's implementation for the Proxy's memory pool manager, but performed significant code refactorings on the Proxy's port process. However, these changes were to the implementations inside each of the states (as well as his own method of implementation externalizations). Thus, the state transition model used by Weeks is identical to Harmon's model. As a result of this continuity between the implementations, the toolkit's externalization of Weeks' Port process implementation is identical in structure (identical UML) to the externalization created for Harmon's model, differing only at the source code level.

4.1.4.8 Queue. Although Weeks maintained the same state transitions for his Port process implementation, he added an entirely new "Queue" process implementation. Figure 4.10 shows the queue process added by Weeks. As in the previous externalizations, the process implementation was externalized using the supporting libraries and the state pattern.

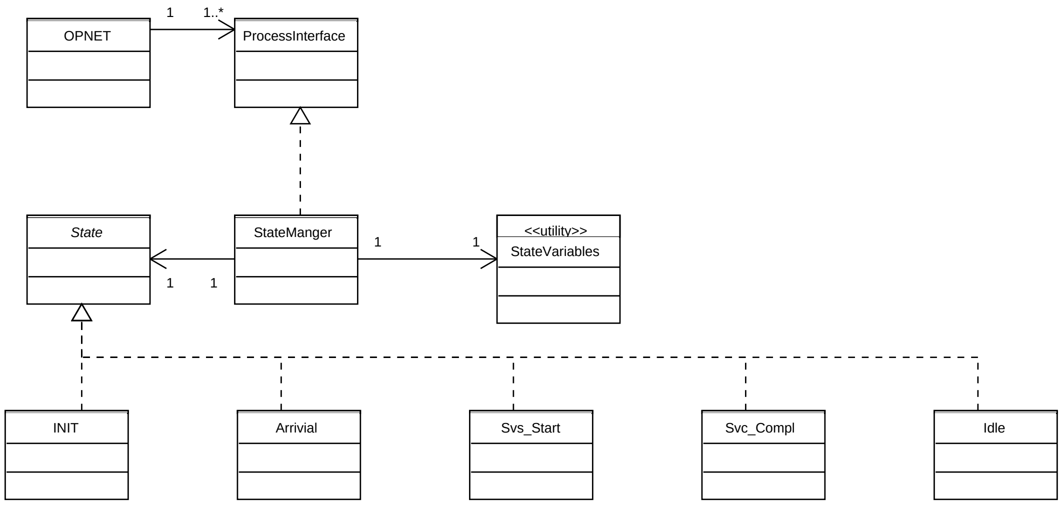

Figure 4.11: Proxy Queue Refactoring UML. This UML represents the externalized version of Weeks' modified version of the OPNET "acb_fifo" queue.

As in all of the previous refactorings, the process implementation is externalized using the methods previously outlined (Figure 4.11).

Thus, we must modify Weeks' proxy port node model (Figure 4.12) with the necessary Esys model interfaces on the pertinent nodes (Figure 4.13). Because the node model specification for the proxy needs to accommodate both buffering implementations, the final toolkit-enabled proxy must contain a superset of all the necessary processes utilized in the implementations. For implementations that do not require use of all of the processes, such as the first buffering strategy, a no-op "do nothing" implementation will be installed by the toolkit. This "do nothing" implementation simply forwards all packets it receives with zero processing time. Thus, this technique allows for the discrepancy in the number of utilized processes between the two implementations without affecting the logic of the implementations or their functioning.

4.1.5 Zero-Code Re-Compilation. One of the major requirements of OPNPT is to be able to add new process implementations, and thus define and use new protocols, without re-compiling any code. Of particular interest is the part of code that maintains references to the available implementations of processes, called the

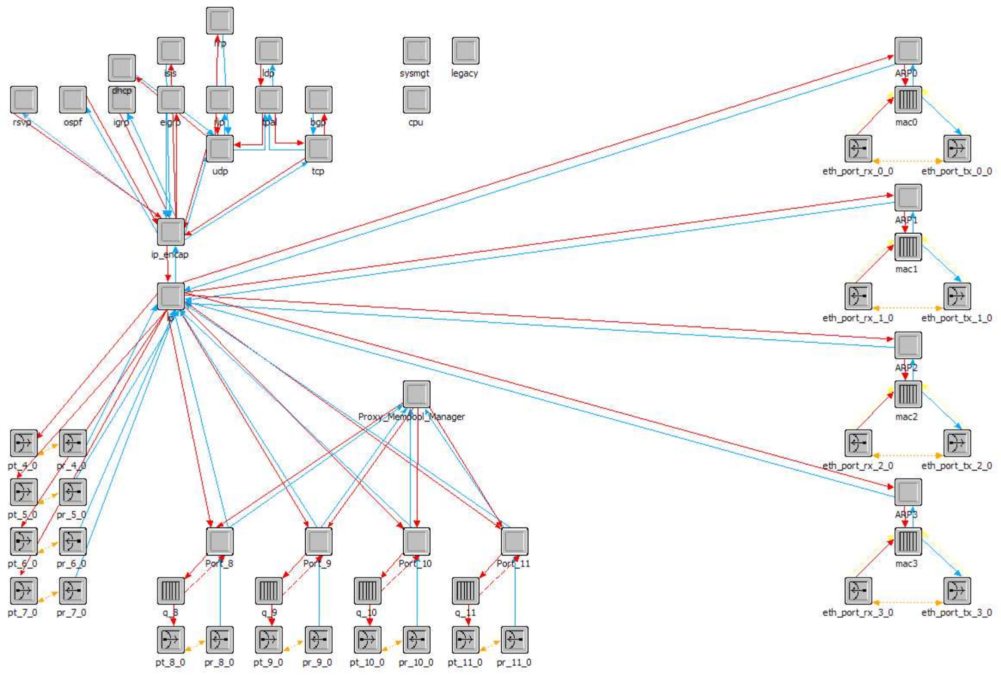

Figure 4.12: Strategy 2's Original Proxy Node Model. This node model is identical to strategy 1's Proxy node model (Figure 4.4), with the exception of the addition of the queueing process between the transceiver and the port process.

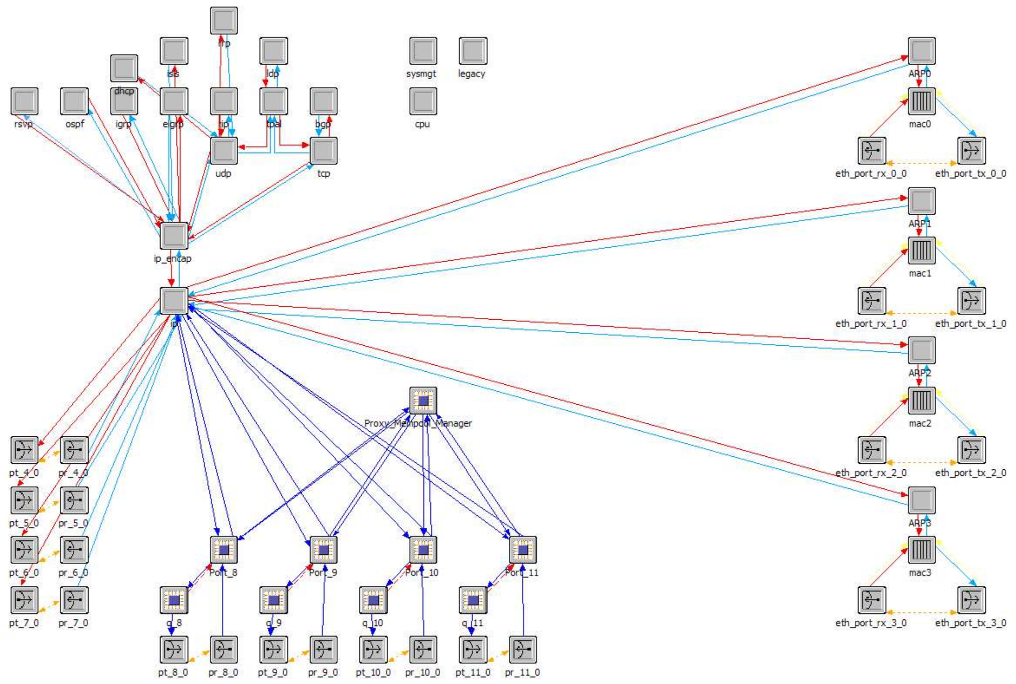

Figure 4.13: Final Proxy Version. This node model shows the final installation of the Esys modules on the pertinent processes. Because Weeks' node model contains an extra process, this node model represents the superset of processes between Harmon and Week's Proxy implementations.

assembler. The addition of a new implementation would require that the assembler be able to instantiate new implementations on the fly at runtime without need for re-compilation. To achieve this functionality, process implementations compiled for use with the toolkit are compiled as dynamically linked libraries (DLL's). This allows the toolkit to leverage the power of the operating system to load new implementations at run-time without need for re-compilation. However, the DLL still needs to provide a mechanism for class instantiation. DLL's can be loaded in two ways under Windows. First, the client can statically link to a library that handles DLL interaction. This has the advantage that the DLL can contain a class and export information about the class for use in the client application. However, the loading of the DLL is handled at program startup via compiler pre-processor directives, and thus deciding at runtime which DLL to dynamically link is not possible. The second method of DLL loading does allow for runtime decisions about which DLL to link against at runtime. However, in the Windows API, the DLL can only export functions as simple function pointers, not the more complicated typing information associated with classes. Java, for example, does allow the dynamic loading of classes at runtime via the class loader. Thus, to circumvent the absence the equivalent of the Java class loader in the C/C++ based Windows API environment, the implementations compiled as DLL's for use with the toolkit export one simple function that returns a reference to the class's underlying static instantiation method (Figure 4.14). Thus, client code can dynamically load a new DLL that contains a particular object oriented implementation, receive a reference to the implementation's instantiation method, and call the method to create a new instantiation of the class. Table 4.1 summarizes linking strategies.

Thus, these three factors combined form a C/C++ based dynamic class loader, similar to that of Java. Unfortunately, this mechanism does not preserve type safety in the dynamically loaded class. Preserving type safety in dynamically linked implementations is an open area of software engineering research [7]. Listing 4.2 shows the DLL `getInstance()` function.

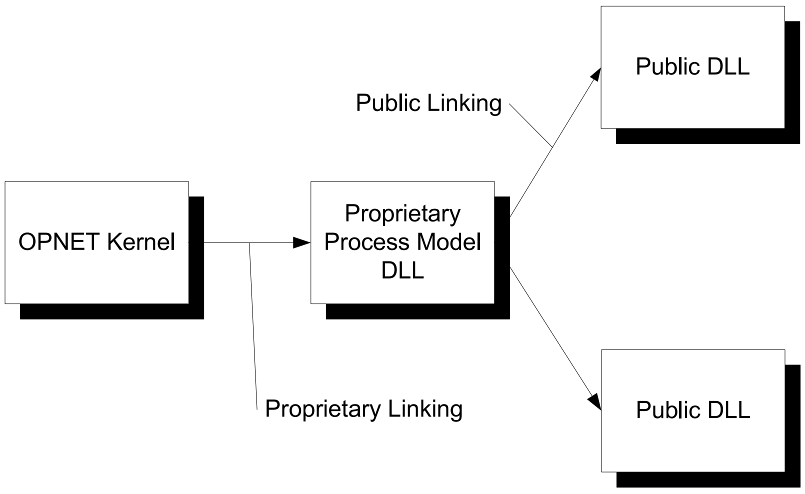

Figure 4.14: Linking Mechanism. The OPNET simulation links to the expected DLL produced by compiling the hook process model. Because this linking mechanism is proprietary, the toolkit's hook process model must, in turn, reference the hot swapped implementation DLL. Thus, this provides a public linking mechanism available and modifiable by clients of the toolkit.

Table 4.1: Linking Strategies. Three linking strategies are in common use today. Compile-time and incremental linking are theoretically equivalent as both linkers have the same information available to them that was available to the compiler. Dynamic linking can add new, previously unknown code to a program at runtime. OPNET utilizes one of the first two strategies; the toolkit utilizes dynamic linking.

Linking Strategies

Type:	Occurrence:	Controlled By:	Used By:
Compile Time	Post-Compile Time	Compilers	OPNET
Incremental	First method/function invocation	OS	OPNET
Dynamic	User Defined	Application/OS	Toolkit

```
#ifdef DLLDIR_EX
 #define DLLDIR __declspec(dllexport) // export DLL information
#else
 #define DLLDIR __declspec(dllimport) // import DLL information
#endif
#include "StateManager.h"
class ProcessInterface;
typedef ProcessInterface* (*GET_NEW_INSTANCE)();


//Export the methods of the class we want available for dynamic
//run-time usage by the DLL client.

extern "C" {
    GET_NEW_INSTANCE DLLDIR getInstance();
}
```

Listing 4.2: DLL Class Loader. This listing shows how the DLL associated with a particular process implementation

```
#include "Stdafx.h"
#include "DLLCode.h"
#include "StateManager.h"
GET_NEW_INSTANCE getInstance(){
    return ((GET_NEW_INSTANCE) StateManager::getInstance);
};
```

Listing 4.3: DLL Implementation. This listing shows the implementation of the DLL function exported in listing 4.2. The function returns a function pointer to the underlying `StateManager`'s instantiation mechanism.

The `getInstance()` implementation returns a function pointer to the `State Manager`'s `getInstance()` method, as shown in Listing 4.3. The `StateManager`'s `getInstance()` implementation thus creates a new instance of the `StateManager` class (and perform other initialization chores if necessary) and returns a pointer to the new instance, as shown in Listing 4.4.

4.1.6 Example. There are two ways to specify the proper configuration and organization of protocol implementations. The first is to drop all process implementation DLL's into a single directory, and place an "implementation_config" XML file that defines protocols and the set of implementations that comprise them. When the

```
StateManager* StateManager::getInstance(){
    return new StateManager();
}
```

Listing 4.4: StateManager Creation Mechanism Implementation. The method simply returns a pointer to a new instantiation of the StateManager class.

```
//make a new scenario
Scenario* scenario1 = new Scenario();
Simulation* sim1 = Simulation::getInstance(scenario1);
//specify implementation directory
sim1->setImplementationDirectory("E:\\Toolkit\\implementations");

//specify other important directories
sim1->setStatisticDirectory("E:\\Test\\Stat");
sim1->setWorkingDirectory("E:\\Test\\Working");

//automatically register protocols based on the config file
sim1->initializeFromXML("implementation_config.xml");

//specify which protocols to use for this simulation run
sim1->useProtocol("PacketDropper");
sim1->useProtocol("ProxyPortDFH");
sim1->useProtocol("ProxyQueue");
```

Listing 4.5: Specifying Protocol Implementations. This code example demonstrates the API for specifying protocol implementations post compile time.

toolkit initializes, it reads the configuration XML file and automatically registers the defined protocols. Once the protocols have been registered, the user specifies that a given protocol be used at the next running of a simulation via the `useProtocol()` method. Additionally, there are several methods in the toolkit API that allow the user to register new protocols themselves programmatically. Listing 4.5 demonstrates these techniques.

4.1.7 Use Case: Running the Properly Configured Simulation. After the simulation's scenario has been specified per use case 1 and the protocols defined and chosen per use case 2, the user then wants to run the simulation. Running the simulation requires specifying a mechanism for traffic generation, controlling the execution of the simulation itself, and collecting the results.

Table 4.2: OP-NPT Simulation Execution API. This table shows the API's available for simulation execution for applications using the OP-NPT libraries.

Simulation API
Method:
runSimulationToCompletion()
runSimulationToTime(double time)
runSimulationOneEvent()

4.1.7.1 Traffic Generation. The OP-NPT supports all of the built-in OPNET applications. Users wishing to use one of the built in applications simply builds a scenario that includes application and profile configuration nodes using the same mechanism as any other node in the simulation. OP-NPT provides application and profile configuration nodes that have all pertinent attributes promoted up to the level of the OPNET command line. This enables the toolkit to manipulate their values external to OPNET. To set the values of the attributes, OP-NPT provides an API to set generic name, value attribute pairs for any object in the simulation. OP-NPT uses these values to generate an OPNET environment file prior to simulation execution. Thus, when the simulation executes, the proper values for all attributes are set. Additionally, the user can choose to handle traffic generation completely on his own using custom implementations of source and sink nodes, for example.

4.1.7.2 Controlling Simulation Execution. Users may control the simulation execution via the API of the `Simulation` object. The API provides for running the simulation to completion as well as other conditions as shown in Table 4.2.

Thus, the user of OP-NPT can harness the full power of C++ when generating simulation execution scripts, instead of using the restrictive OPNET GUI, or hierarchies of Windows batch files which can quickly become unmanageable. For example, Listing 4.6 executes a simulation 30 times. This is also easily accomplished with a batch file or through the OPNET GUI. What is far more easily accomplished with

```
for(int i = 0;i < 30; i++){
sim1->runSimulationToCompletion();
}
```

Listing 4.6: Batching Simulations Programmatically. Running simulations can be achieved with

```
for(int i = 0; i < 5; i++)
}
  sim1->useProtocol(protocolArray[i]);
     for(int j = 0;j < 30; j++)
     {
          sim1->runSimulationToCompletion();
     }
}
```

Listing 4.7: Batching Protocol Implementations. This code example demonstrates the API for specifying multiple protocol implementations over many simulation runs.

OP-NPT's programmatic interface is more interesting simulation sequences, such as Listing 4.7, which runs a simulation 30 times on each of 5 different protocols under evaluation.

And because OP-NPT's use of the dependency injection framework prevents the re-building of OPNET network objects when different protocols are swapped in and out, the swapping mechanism is instantaneous, and thus suitable for use inside a loop, as in the previous example.

4.1.7.3 UC4: Collecting Results. The OP-NPT handles the collection of results via two methods. First, customized link and wireless models record the sending and receipt of all packets. OP-NPT then stores this as raw data in a SQL-lite database. Thus, the data can then be exported for statistic calculation, or various SQL queries can be run on the data to calculate statistics with the database. The second method of statistic collection capitalizes on OPNET's built in statistic collection abilities. The toolkit can specify which previously registered statistics should be collected before simulation execution via a user created statistic probe. After simulation execution, the OPNET readable (not human-readable) statistic file can be opened with the OPNET GUI, and various graphs and reports generated.

4.1.8 Use Case: Interfacing with OP-NPT from Java. The OP-NPT libraries provide a Java interface for the most utilized API methods. Thus, in addition to C/C++ programmatic control of OPNET simulations, developers from the Java community (such as concurrent research efforts in network visualization) can design, run, and collect data from OP-NPT utilizing a native Java interface.

4.1.9 OP-NPT Conclusion. OP-NPT utilizes the toolkit libraries to provide a programmatic interface for OPNET scenario generation, selection of specific implementations of NCW algorithms under research, and management of simulation results. Additionally, because of the dynamic linking techniques utilized, users can add new implementations for components without re-compiling any code.

4.2 OPNET-Unit

4.2.1 Introduction. Many OPNET developers already use small, isolated node models to test process model implementations. However, these testing solutions provide for only the specific case, with a given process model implementation in mind. Testing additional process models would require creating new node models, or editing the existing one. Testing of multiple test cases on a process model would require defining a separate input generation process model for every type of input needed in testing. Moreover, managing the batch running of multiple test cases would require batch execution of multiple simulations. This testing technique quickly becomes unmanageable. Testing in the general case, i.e. testing any given process model with minimal OPNET coupling, requires the ability to "inject" the node under test with a process model implementation post-compile time.

4.2.2 Overall Architecture. The overall architecture (Figure 4.15) of OPNET-Unit consists of three distinct tiers: 1) The specialized node model which executes in

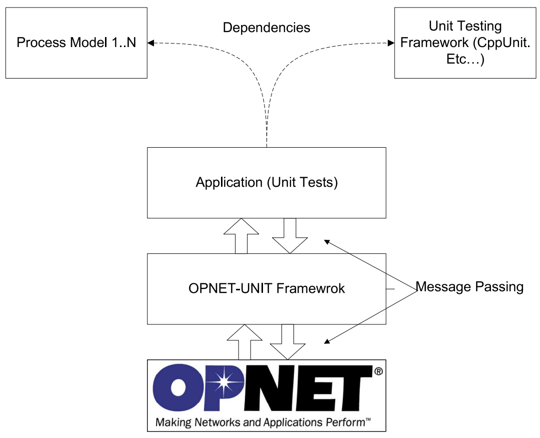

Figure 4.15: Overall OPNET-Unit Architecture. The architecture consists of three tiers: the OPNET node model, the OPNET co-simulation, and the actual unit tests as implemented by the user.

the OPNET simulation, 2) the OPNET Co-Simulation, and 3) the unit tests themselves. The Co-Simulation defines the entry point for the application, and thus uses the OPNET simulation platform as an external library. The top application level contains the actual unit tests the user wishes to execute on the underlying process model, and uses two libraries: the OPNET-Unit framework to interact with the simulation, and the actual unit testing library of the user's choosing such as CppUnit or CxxTest to enforce assertions and collect test results.

4.2.3 OPNET-Unit Node Models. As described in previous sections, modules called node models define high level objects participating in an OPNET simulation such as routers. The OPNET-Unit test environment consists of only one generic high level node which contains a specialized node model, best suited for testing a given process model. However, the processes in this node model communicate via OPNET's Esys interface to the OPNET-Unit framework which makes important decisions about their operation. Thus, an inspection of the node model reveals little about its intent. To gain a better conceptual understanding of the intent of OPNET-Unit's specialized node model, I describe an intent revealing "conceptual" node model first, and then the actual node model.

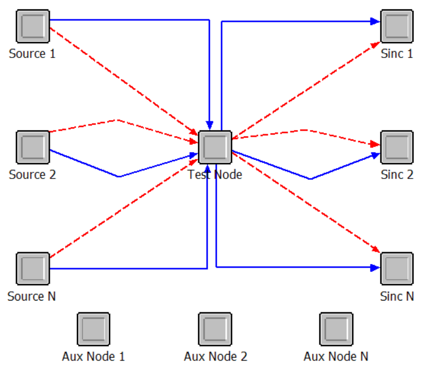

Figure 4.16: OPNET-Unit Conceptual Node Model. This node model serves as a communication tool to convey the intent of the actual node model in use by the framework, figure 4.17

4.2.4 Conceptual Model. The conceptual node model consists of four classes of nodes: source nodes, sink nodes, “auxiliary” nodes, and the testing node (Figure 4.16). The source nodes connect to the test node via both packet streams and statistic wires, and the test node connects to the sink nodes in the same manner. There are anywhere from 0 to N source and sink nodes (limited only by OPNET’s upper bound)[1]. Additionally there are anywhere from 0 to N “auxiliary” nodes which represent all other nodes in the node model which do not possess a direct connection with the node under test. These nodes come into play if the node under test needs to communicate with another node in the node model through which it is not directly connected, via a remote interrupt for example. The node under test can still communicate via remote interrupts with nodes for which it possesses a packet stream or statistic wire connection as well.

From the perspective of a user writing unit tests for the node under test, this is the node model environment. Through this conceptual model, the test environment

[1]Because of the limitations of OPNET kernel procedures, only the first 10 source and destination nodes may have statistic wires. All additional nodes will have only packet stream communication with the node under test.

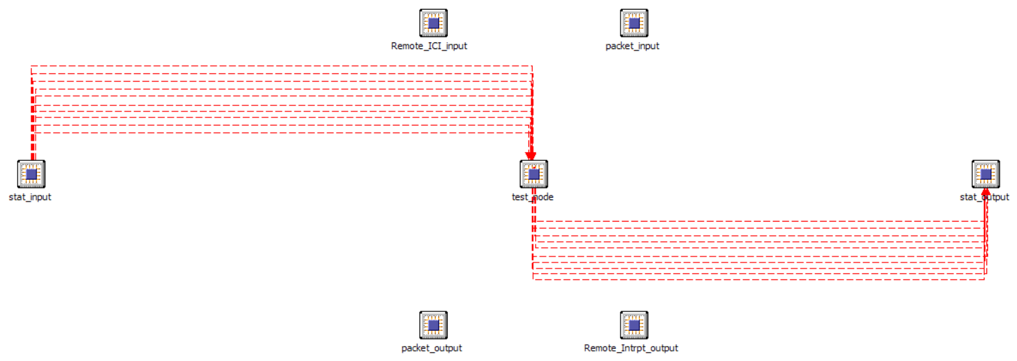

Figure 4.17: OPNET-Unit Actual Node Model. This node model takes extensive advantage of the OPNET Esys interface specification to communicate with the testing framework for pertinent information.

can re-create all possible effects on the node under test that an actual simulation execution could produce.

4.2.5 Actual Model.

OPNET-Unit's actual node model differs substantially from the conceptual node model previously described (Figure 4.17). All nodes in the actual node model utilize the OPNET Esys (External System) interface and specialized process models define them. Through these mechanisms, the underlying OPNET simulation can communicate with the OPNET-Unit framework and vice versa.

4.2.5.1 Packet Streams.

Three source nodes (a packet source node, a statistic source node, and an ICI source node) provide "entrance points" for the testing framework to initiate stream, stat, and ICI interrupts respectively in to the node model via the Esys interface. Interestingly, the packet source node need not be connected to the node under test with actual packet streams because of the `op_pk_deliver()` kernel procedure which allows the packet source process to trigger packet arrival interrupts on arbitrary input streams to the node under test. This ability allows the conceptual model to possess anywhere from 0 to N packet source nodes. However, this is not the case with statistic interrupts.

4.2.5.2 Statistic Wires. In OPNET, communication between nodes via statistic wires requires that each statistic wire be configured to carry a statistic that has been defined a priori in a statistic registering system. Thus, OPNET-Unit pre-defines 25 generic statistic wires, placing an artificial upper bound on the total number of statistic wires available for testing.

4.2.5.3 Test Node. The single testing node possesses the process model under unit testing. The node under test processes the incoming interrupts as it normally would, forwarding packets to whatever output stream it normally would, generating statistics on the static wires it normally would, and generating any remote interrupts as it normally would. These three types of outputs, packet stream, statistic, and remote ICI's, are all directed via the partial mock object (described in the next section) to the packet stream, statistic, and remote ICI listening nodes respectively. These listening nodes, in turn, notify the OPNET-Unit Framework that an event of importance has occurred so the framework can take appropriate action.

4.2.6 A Partial Mock Object.

In order to eliminate changing the process model implementation under testing specifically for the purpose of testing–usually the sign of poor testing practices–some kernel procedures used by the process model under test are re-defined in terms of a partial mock object that provides test specific implementations [13] [14] [21] . This is a partial mock object because the intent is not to provide test oriented implementations for all of the OPNET kernel procedures, which would be complicated and unmanageable, but to provide test-specific implementations for some kernel procedures and default to the original implementations for the majority of the procedures. This enables the implementation under test to execute using the standard OPNET interfaces. Examples of kernel procedures that must be redefined for purposes of testing are procedures that require direct information about the surrounding node model to the process model implementation under test, such as `op_pk_deliver()` which requires the implementation to supply the object ID of the destination node. The re-defined ver-

```
//headerfile
#include <opnet.h>
#include "OPMockObject.h"
#define op_pk_send(pkptr,outstrm_index)
     OPMockObject::mockPKSend(pkptr,outstrm_index)
```

Listing 4.8: Kernel Procedure Redefinition. This example shows the redefinition of the kernel procedure `op_pk_send()` to use the definition provided by the mock object.

sion of this kernel procedure steers all packets sent with `op_pk_deliver()` to the unit testing's specialized node model's packet destination node. Other examples of kernel procedures that require re-definition include `op_intrpt_force_remote()`, `op_intrpt_schedule_remote()`, `op_ima_obj_attr_get()`.

Thus, the only change developers must make is to use a "unit testing" opnet.h include file rather than the regular OPNET include file. This unit testing include file contains the kernel procedure redefinitions, which define the kernel procedures in terms of the corresponding methods of the mock object, instead of the usual "prim" functions that ship with OPNET.

OPNET-Unit provides a basic set of kernel procedure re-definitions, suitable for handling stream, stat, and remote interrupts generated by the node under test. Additionally, OPNET-Unit provides re-definitions to allow the testing framework to supply process model attributes and set-up statistic wire registrations. Users wishing to unit test more complicated processes will want to further re-define more kernel procedures and expand the partial mock object library to their particular use case. Because of OPNET's procedural style of development, users cannot define new kernel procedures by sub classing the OPNET kernel and overriding certain kernel procedures, rather, they must add new pre-compiler directives, overriding the old symbol definitions with the new behavior, as shown in Listing 4.8.

Thus, once developers re-define pertinent kernel procedures with the appropriate mock object method calls, the state of the mock object (and thus the result of executing the re-defined kernel procedures) can be controlled from the unit testing code in the application layer. An example of this is the re-definition of the kernel procedure, `op_ima_attribute_get()` which returns the value of a process model attribute of any object, including its self. Many process models commonly use this kernel procedure during the initialization stage of many process models to initialize operating parameters, such as queue size, time-out settings, and other configuration options. With the kernel procedure re-defined in terms of the mock object, the unit testing script can pre-set any value for any arbitrary attribute for testing purposes, as well as modify these values during or between tests as testing requirements dictate.

4.2.7 Executing Customized Test Simulations with Precision Control.

Utilizing the OPNET External System Access (ESA) API, the OPNET-Unit Framework controls the execution of the underlying OPNET simulation with great precision—down to the event when necessary. Alternatively, the OPNET-Unit framework makes use of the "observer" design pattern to only interrupt the executing simulation when events of interest occur. When the framework initializes the simulation, it registers several call-back methods with the specialized process models located in the listener nodes of the simulation's node model. This way, the simulation can run uninterrupted, notifying the framework that the node under test has sent a packet, modified a statistic, or initiated a remote interrupt when that particular event has occurred and can then decide to check the state of the node under test or continue execution. Thus, the framework can execute the simulation event by event, checking assertions all along the way, or execute the simulation until key events, only then pausing the simulation to check assertions. The framework also recognizes other key events to pause the simulation and check assertions, as listed in Table 4.3.

4.2.8 Observing Simulation State.

In testing a process with OPNET-Unit, there are two important parts of the simulation that should be observed for testing purposes. First, the state of the actual process model defining the node under test. Second, the state of the surrounding simulation (other nodes, packet interrupts, statistics) that the node under test might have altered. Observing the state of the node

Table 4.3: OPNET-Unit Simulation Execution API. This table shows the API related to the control of simulation execution. Modeler offers `incrementUntilOneEvent()` out of the box, however the application layer must implement the remainder of the API.

Precision Execution API

Method:
incrementUntilOneEvent()
incrementUntilPacketArrival()
incrementUntilPacketArrivalOnStrm()
incrementUntilStatArrival()
incrementUntilStatArrivalOnStrm()
incrementUntilRemoteInterrupt()
incrementUntilUnforcedStateReached()
incrementUntilUnforcedStateLeft()

under test is largely a matter of the way the tester implemented the process model which defines the node under test. However, using the OPNET-Unit state pattern refactoring technique previously discussed, observing such aspects of the node under test as the status of all the state variables, and the actual state the model is currently in (via the state transitions) is straightforward, requiring simple method calls. Observing the state of the surrounding simulation is possible using methods supplied by the framework. OPNET-Unit's specialized node model reports pertinent information such as packet arrival, statistic output, and ICI pointers back to the framework's published interface for access and use from a standard unit-testing library. Using these methods, a test scenario is possible that sends a packet to the node under test, and then verifies that the node under test then forwards the packet on a given output stream, and writes a certain statistic on a given statistic wire.

4.2.9 OPNET-Unit Conclusion. Effective unit testing has been shown to reduce debugging times for several reasons, most notably through defect localization. Utilization of unit testing in the development process frequently distinguishes advanced programmers from novices [17]. However, in the OPNET simulation environment, unit testing via traditional unit testing methods quickly becomes infeasible and unmanageable because of OPNET use of proto-C, process model paradigms, and heavy reliance on the OPNET simulation kernel. OPNET-Unit proposes a methodology and framework for overcoming each of these difficulties by providing access to process model variables and states, and surrounding simulation state through a clearly defined interface, suitable for testing in a traditional C++ development environment using a xUnit testing framework. With the support of the new framework, new OPNET development methodologies are now possible including TDD (Test Driven Development) [17] and agile forms of development. Moreover, batteries of unit tests could ship with the accompanying process models and serve as robust regression tests, facilitating future OPNET modeler development; researchers seeking to modify an existing protocol could run the battery of tests after each modification of the source code, clearly identifying which specifications the modification no longer fulfills. Lastly, research efforts can improve, as developers implement new protocol designs for use with the OPNET simulation platform, OPNET-Unit can provide a greater degree of confidence that their implementations are true to their original protocol designs, yielding higher quality and more meaningful results.

4.3 Conclusion

This chapter discussed in detail the utilization of the toolkit developed in Chapter 3 into two example applications: The OPNET Network Protocol Testbed (OPNPT), and OPNET-Unit, an OPNET unit testing framework. The chapter discussed the necessary modifications to Maj. Harmon's buffering proxy router node models. Additionally, the chapter discussed the inclusion Harmon's and intern Matt Week's process implementations into externalized DLL's, suitable for use with the framework.

The chapter also discussed the specialized node models used in conjunction with the underlying dependency injection framework discussed in Chapter 3 for the implementation of OPNET-Unit. The chapter shows how the framework can be used to build and execute small test simulations in the context of an xUnit based framework. This framework can then, in turn, become an invaluable tool in the automation of testing suits for process implementations.

V. Application and Analysis of Results

This chapter demonstrates the robust features of the implemented applications discussed in chapter 4, to include mobile scenarios, wireless scenarios, scenario generation from XML source files, hot swappable implementations, and Java interface capabilities. Moreover, the chapter presents an in depth analysis of correctness of the process implementation externalizations discussed in chapter 4. This analysis is performed by running simulations with the original process implementations, re-running the same simulation with the framework, and observing the identical results. Lastly, the chapter provides an analysis of correctness of the OPNET-Unit implementation.

5.1 OP-NPT

5.1.1 Scenario Generation.

5.1.1.1 Generation from XML.

The effect of the builder on the programmatic generation of OPNET readable scenario files is astounding. Without the use of the library, creating even the simplest of scenarios requires 10's of thousands of lines of hard-to-read OPNET API calls. If a developer utilizes the toolkit scenario generation libraries and a simple XML parser, the developer can generate the same scenario from an XML persistence of just 30 lines. Moreover, because the toolkit library API is simple and programmatic, a clever developer could create software that generated 1,000's of different OPNET readable scenario files that followed certain guidelines (constraints on various types of nodes, preference to certain configuration styles), or even use the results of a simulation run to automatically affect the design of the next simulation scenario.

Listing 5.1 shows the format of the XML format readable by the built-in XML parser for the library. Users are free to utilize their own XML parser, toolkit builder library, and XML format to construct their own scenario generation mechanism. For convenience, the libraries provide for this format by default. Listing 5.2 shows an example XML file and figure 5.1 shows the resulting OPNET scenario.

```
<Root>
    <Nodes>
        <Node name = "<node name>" type = "<model name>" numTx = "<# of
            transceivers>" numRx = "<# of receivers>" txName = "<naming
            convention>" rxName = "<naming convention>" xPos = "<x
            coordinate>" yPos = "<y coordinate>">
        <attribute name = "<attribute name>" value = "<value of
            attribute>"/>
        </Node>
    </Nodes>
    <Links>
        <NewLink name = "<link name>" type "<link model>" node1 = "<node
             1 of the link>" node2 = "<ndoe 2 of the      link>">
        <attribute name = "<attribute name>" value = "<value of
            attribute>"/>
        </NewLink>
    </Links>
    <Simulation>
        <SimulationAttributes>
        <attribute name = "<attribute name>" value = "<value of
            attribute>"/>
        </SimulationAttributes>
    </Simulation>
</Root>
```

Listing 5.1: Toolkit XML Format. This Listing shows the default XML format readable by the toolkit libraries. Users are required to specify the number of receivers and transceivers for proper scenario generation. The naming convention allows the building mechanism to properly link nodes in the OPNET environment.

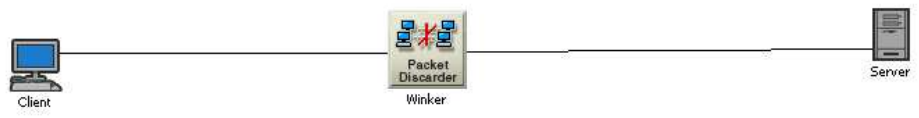

Figure 5.1: Basic OPNET scenario. This scenario contains a client and a server connected by an intermediate packet dropping mechanism.

```
<Root>

<Nodes>
<Node name = "Client" type= "mc_ppp_wkstn_adv" mobile = "Yes" numTx = "1
    " numRx = "1" txName = "ip_tx_X_0" rxName = "ip_rx_X_0">
</Node>
<Node name = "Server" type= "ppp_server_adv" mobile = "Yes" numTx = "1"
    numRx = "1" txName = "ip_tx_X_0" rxName = "ip_rx_X_0">
</Node>
</Nodes>
<Links>
<NewLink name = "link1" type= "ppp_adv" node1 = "Client" node2 = "Server
    "/>
</Links>

<Simulation>
<SimulationAttributes>
<attribute name = "duration" value = "3600"/>
</SimulationAttributes>
</Simulation>

</Root>
```

Listing 5.2: Example Scenario. This listing shows an example XML representation of the scenario depicted in figure 5.1.

```
//Get the standard scenario builder
ScenarioBuilder* builder = new StandardScenarioBuilder();


//add two nodes
builder->addFixedNode(1,1,-30,60,0,"Client","mc_ppp_wkstn_adv5","
    ip_tx_X_0","ip_rx_X_0");
builder->addFixedNode(1,1,15,60,0,"Server","mc_ppp_server_adv5","
    ip_tx_X_0","ip_rx_X_0");
builder->addFixedNode(2,2,0,60,0,"Winker","
    dfh_packet_discarder_node_model_28Dec","output_X","input_X");

//link them together
builder->addLink("link1","ppp_adv","Client","Winker");
builder->addLink("link1","ppp_adv","Winker","Server");

//add the application "nodes"
builder->addFixedNode(0,0,50,50,0,"Profile_Config","
    mc_toolkit_gna_profile_config_adv","","");
builder->addFixedNode(0,0,40,50,0,"ApplicationConfig","
    mc_toolkit_gna_attrib_definer_adv","","");

//add the statistic communication node
builder->addFixedNode(0,0,30,50,0,"Stat_Comm","mc_stat_comm","","");
```

Listing 5.3: Programmatic Scenario Generation. This listing shows the use of the toolkit's C/C++ libraries to generate a scenario programmatically.

5.1.1.2 Generation from C++ API. Additionally, simple programs can programmatically generate scenarios, as shown in listing 5.3. Although OPNET's EMA libraries fundamentally provide the same functionality, the toolkit libraries are more accessible, and greatly reduce the time, effort, and complexities associated with scenario generation. Additionally, the libraries provide new functionality not found in OPNET when combined with the toolkits' programmatic raw data collection mechanisms. With these two functionalities combined, users can author applications that automatically adjust the setup and configuration of the next batch of simulation runs based on the results of the last batch.

5.1.2 Mobile/Wireless Scenarios. Because OP-NPT does not make any assumptions about the types of nodes a user may want to insert into a given scenario,

```
<Root>
<Nodes>
<Node name = "Client" type= "mc_wlan_wkstn_adv"
        numTx = "1" numRx = "1" txName = "ip_tx_X_0" rxName = "ip_rx_X_0
           "/>
<Node name = "Server" type= "mc_wlan_server_adv"
        numTx = "1" numRx = "1" txName = "ip_tx_X_0" rxName = "ip_rx_X_0
           "/>
<Node name = "Profile_Config" type= "mc_gna_profile_config_adv"
        numTx = "0" numRx = "0" txName = "" rxName = ""/>
<Node name = "App_Config" type= "mc_gna_attrib_definer_adv"
        numTx = "0" numRx = "0" txName = "" rxName = ""/>
</Nodes>
<PacketDiscarder type = "dfh_packet_discarder_node_model_28Dec"/>
<Links>
<!--No links needed for wireless network!!-->
</Links>
</Root>
```

Listing 5.4: XML Representation of Mobile Scenario. This XML shows the representation of a toolkit compatible mobile scenario. The toolkit intern reads this XML file and makes the appropriate toolkit API calls.

OP-NPT automatically supports mobile nodes in mobile scenarios as well as wireless nodes without any extra programming on the part of the user. Listing 5.4 shows the generation of a mobile scenario.

5.1.3 Java Interfaces. OP-NPT utilizes the native interface capabilities of Java to create a simplified Java API for the existing C++ based API. This allows Java applications, such as NetViz, to integrate with the toolkit libraries and take full advantage of their capabilities. Listing 5.5 builds and executes the same simulation shown in Figure 5.1.

5.1.4 Verification of OP-NPT Externalizations. Since the previous research efforts did not have access to OPNET-Unit or any other similar unit testing tool, there are no regression unit tests to run on the externalized versions of the implementation externalizations. Thus, in order to perform verification on the externalized implementations, we must re-create previously run simulations and compare the results.

```java
public static void main(String[] args) {
    Simulation sim = new Simulation();
    sim.generateScenario("test");
    sim.setImplementationDirectory("E:\\Toolkit\\implementations");
    sim.initializeFromXML("implementation_config.xml");
    sim.useProtocol("ProxyMempoolManager");
    sim.useProtocol("ProxyPortDFH");
    SimulationExecutor testThread = new SimulationExecutor(sim);
    testThread.start();
  }
```

Listing 5.5: Java Toolkit API Example. This example demonstrates the invocation of the toolkit's Java API for use with Java applications The API offers a simplified subset o the functionalities available in the original C++ API.

If we receive similar results for a variety of different simulation configurations, then we can be reasonably confident that the re-factored version of the buffering protocols are equivalent to the original.

5.1.4.1 Setup of Experiments. The verification simulations in this section were conducted in the following manner. For each simulation, a control simulation was constructed manually in OPNET utilizing the standard OPNET GUI options. Next, the simulation was duplicated via an automated OPNET option. In the duplicated scenario, the original non-toolkit network objects were manually replaced with the toolkit enabled network objects. The duplicated scenario was then run via the toolkit. Although the toolkit has the ability to build the simulations programmatically, the duplication technique preserves all settings from one simulation to the next and thus eliminates unwanted variability.

5.1.4.2 Verification 1: The first verification simulation set verified the "packet dropper" externalization. First, a control simulation was run that consisted of a client and a server without a winking link. Then, experiments with various settings and random number seeds were conducted with both the original implementation,

Table 5.1: Control Simulation. This table shows the response time of the download of a 20mb file via the file transfer protocol over an unchallenged link. This simulation was conducted via the standard OPNET GUI.

Control

Seed:	Response Time (Sec.)
128	22.731632879988
256	22.731632879988
512	22.731632879988
768	22.731632879988
1024	22.731632879988

Table 5.2: Simulation Settings. This table shows the settings for the packet dropper used for the first simulation set.

Application Settings: Set 1

Setting:	Value:
Inter-Wink Interval	.1
Failure Probability	.3

and the toolkit's version of the implementation. The results are summarized Table 5.1.

This control simulation sequence demonstrates several important points. First, the response time for the file download remains constant despite changes in the random number seed. This is attributable to the configuration of the application settings in OPNET to constant values, instead of values based on a random distribution. This is an important fact to note, because it isolates future observed changes in response times with respect to the random seed to the implementations under observation, and not the testing set-up and application configuration settings.

This battery of tests shows that a simulation run utilizing the toolkit to dynamically hot swap an implementation behaves identically to the original implementation. This verifies several key pieces of the toolkit methodology:

- The externalization (refactoring) mechanism of a given implementation does not change it's behavior with respect to interaction with the simulation kernel.

Table 5.3: Packet Dropper Verification Results. This table shows that the toolkit implementation behaved identically to the original implementation over a variety of seeds.

Packet Dropper Verification Results

Seed	Expected	Observed	Delta
128	105.2296836	105.2296836	0
256	38.331074559999	38.331074559999	0
512	54.130632239994	54.130632239994	0
768	104.730596079999	104.730596079999	0
1024	69.930492079998	69.930492079998	0

Table 5.4: Verification 2 Settings. This table shows the settings used for the second verification simulation.

Application Settings: Set 1

Setting:	Value:
Inter-Wink Interval	0.1
Failure Probability	0.01

- The toolkit's mechanism for dynamically linking and injecting dependencies into an executing simulation does not affect the behavior of the externalized implementation.
- Statistics can be effectively collected for analysis.

5.1.4.3 Verification 2: The second verification set compares the response time of strategic buffering version 1 with the same statistic gathered from executing the simulation with the toolkit. As in previous simulations, multiple simulations over several seeds were analyzed. Table 5.5 summarizes the results.

Figure 5.2 charts the differences between the original strategic buffering version 1 implementation and the toolkit's hot swappable implementation. Interestingly, the toolkit's results varied from the expected values in a non-uniform manner. There is no correlation between seed values and the delta between the observed and expected values.

Table 5.5: Strategic Buffering Version 1 Verification Results. This table shows the relationship of the original strategic buffering mechanism with the toolkit implementation. The toolkit implementation's response time varies from the expected value with respect to the seed utilized by the OPNET random number generator. The error between the observed and expected values varied from as much as $3.05711E-07$ to as little as $1.9235E-08$.

Strategic Buffering Version 1 Response Time

Seed	Strategy 1	Toolkit	Delta
128	24.149466047411	24.149466028176	1.9235E-08
256	24.191545973788	24.191545938771	3.5017E-08
512	24.098585924821	24.098586198175	-2.73354E-07
768	24.301650322633	24.301650399049	-7.6416E-08
1024	24.191546146677	24.191545840966	3.05711E-07

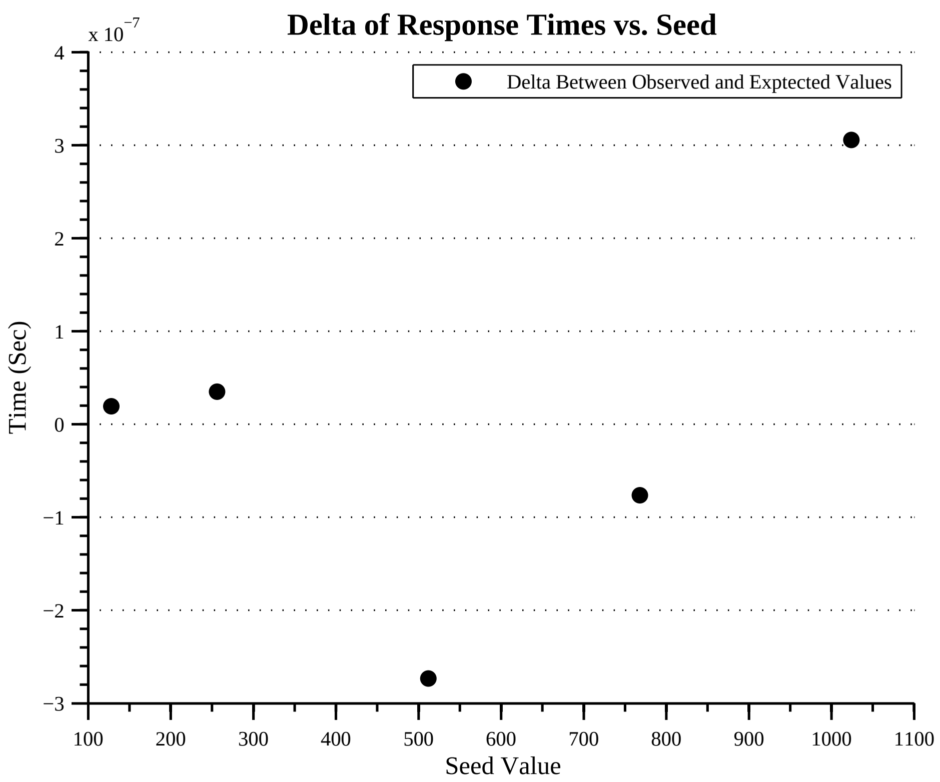

Figure 5.2: Delta of Response Times Between Toolkit Strategic Buffering Version 1 and Expected Values. This chart shows the delta's between the original strategic buffering version 1 (expected) and the toolkit's hot swapped version (observed). The differences are practically zero.

Table 5.6: Verification 2 Settings-Second Run. This table shows the settings used for the second run of the second verification simulation.

Application Settings: Set 2

Setting:	Value:
Inter-Wink Interval	0.1
Failure Probability	0.3

Table 5.7: Verification 2 Results. This table shows the relationship between the original strategic buffering mechanism and the hot swappable toolkit version. This test run is similar to that shown in table 5.5, however, the link's probability of failure has been increased from 10% to 30%, thus increasing the burden on the buffering mechanism's implementation.

Strategic Buffering Version 1 (Response Time)

Seed:	Strategy 1:	Toolkit:	Delta:
128	25.426082457928	25.426082387351	7.0577E-08
256	25.48301636755	25.483016415367	-4.7817E-08
512	24.882090987013	24.882090802223	1.8479E-07
768	25.368778052944	25.368778310676	-2.57732E-07
1024	24.888534344855	24.888534083832	2.61023E-07

The experiments prior to this eliminate some reasons for the discrepancy. First, the seed values change the response time by more than the delta between the observed and expected values. This eliminates the random number generation process itself from causing the difference. Second, the swapping mechanism has been shown to produce identical results in prior experiments, so it is unlikely to be the cause in this experiment. This leaves three options left as the cause:

1. Despite best efforts, the simulation settings were not consistent between the control simulation and the toolkit simulation.
2. The toolkit's swappable implementation is not the same as the original. The refactoring process introduced a defect.
3. Rounding and truncation errors in the statistic collection mechanism.

Table 5.8: Disabled Verification Settings. This table shows the response time of the scenario with the router's buffering mechanism disabled via its own disabling mechanism. The identical results support the hypothesis that the toolkit's buffering mechanism contains a defect. Additionally, a "no-operation" implementation of the key processes was implemented that simply forwards all packets. The results from the no-op implementations were also equivalent to the disabled setting of the original implementation.

Disabled Buffering

Seed:	Version1:	Toolkit Disabled:	Delta
1024	116.640850319997	116.640850319997	0

Another batch of simulations were run. This time, the link's probability of failure is 30%. This second test shows similar discrepancies (Table 5.7). Next, we compare the results of running the same simulations with the buffering mechanism of the routers disabled. Table 5.8 shows the results.

5.1.5 Verification 3. Next, we verify the second strategic buffering mechanism. Figure 5.3 shows the relationship between the original implementation and the toolkit. Delta's between the observed and expected values varied from as close as .007 seconds to as large as 4.317 seconds. As in the verification of the first strategic buffering version, the errors do not appear to be systematic–the delta varies unpredictably from seed to seed.

5.1.6 Verification 4. In the last verification simulation set for OP-NPT, we use the node model for the router used in buffering strategy 2, and install the implementations for buffering strategy 1. For the remaining process that requires no implementation, we install the no-op implementation that simply forwards packets with zero processing time. Discrepancies between the observed and expected values are shown in Figure 5.4.

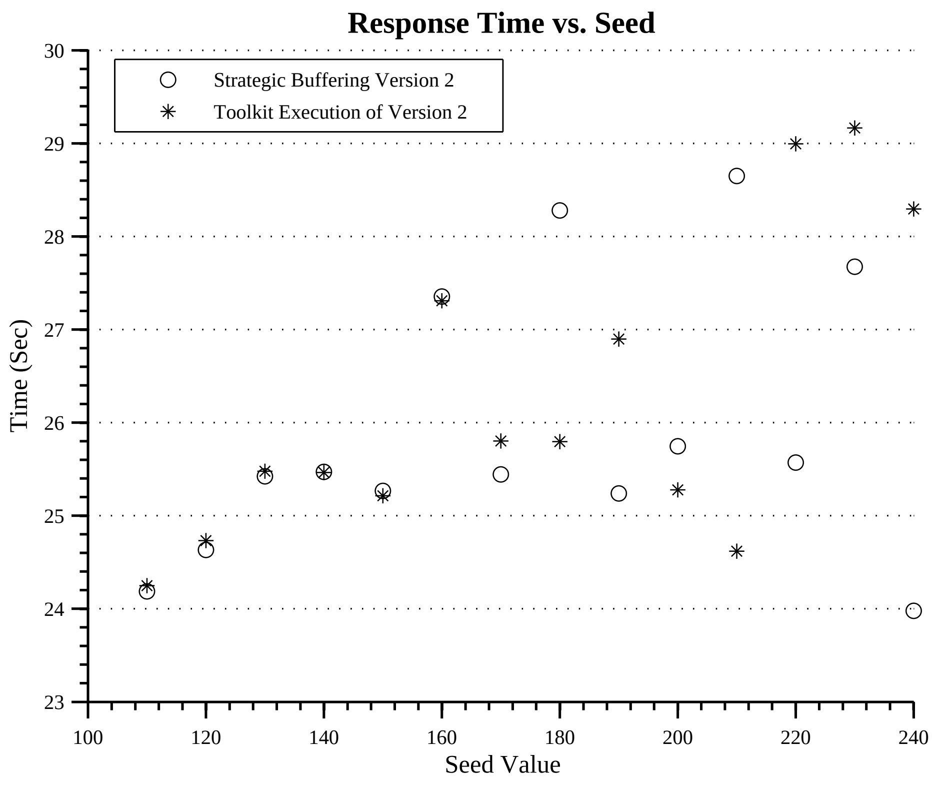

Figure 5.3: Strategic buffering 2 vs. Toolkit Implementation. This graph shows the relationship between the results of the original implementation of the second strategic buffering mechanism and the toolkit's execution of the same mechanism. The graph reveals that although some data points between the two versions are similar, many are quite different. This is evidence of the original mechanism being fundamentally altered during the toolkit installation process.

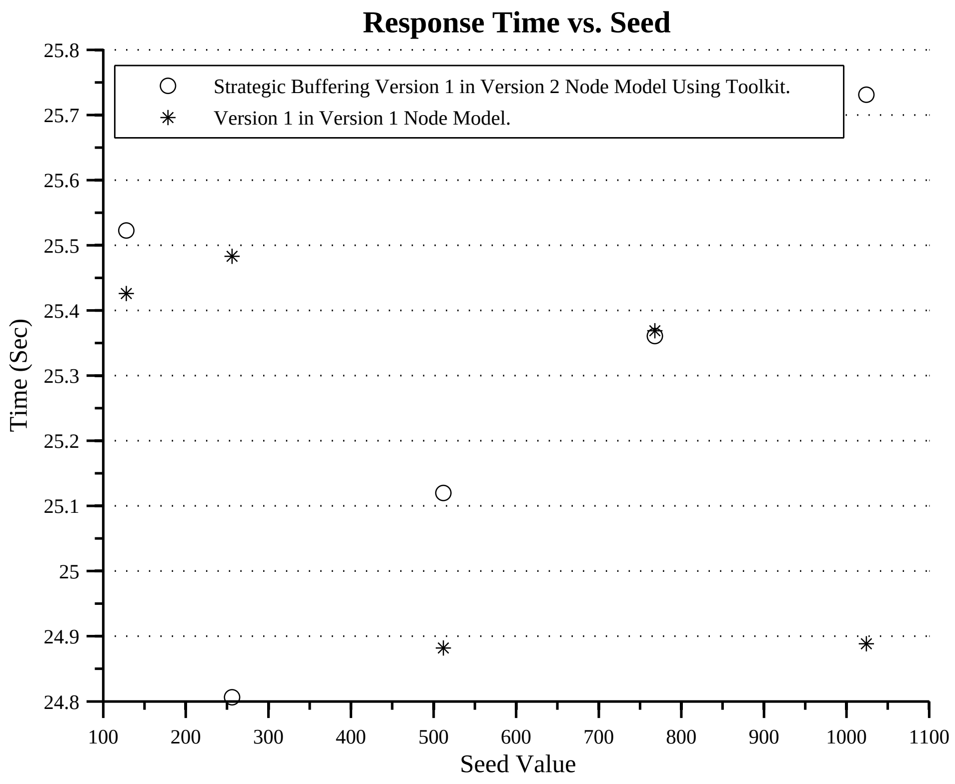

Figure 5.4: Strategic buffering 1 vs. Toolkit Implementation. This graph shows the relationship between the results of the original implementation of the first strategic buffering mechanism integrated into the second buffering mechanisms's node model. The graph reveals variance between the two models on the order of less than a second.

Table 5.9: Verification Result Averages. This table shows the average delta's between the three different verification cases. The most simple case performed the best with no difference between observed and expected values. More complicated multi-process implementations performed well in some cases, but less well in others. Inter-process verification results were similar.

Strategic Buffering Averages (Response Time)

Verification Case	Average Delta (sec)
Single Process	0
Multi Process-version 1	1.41947E-07
Multi Process-version 2	1.25
Inter Process-version 1	0.09

5.2 OP-NPT Results Conclusion

The testing and verification of OP-NPT included the testing of five process implementations, swapped in and out in different combinations in a total of three distinct node models. The verification process included the following test cases of swappability:

- Swapping single process implementations.
- Swapping multiple process implementations.
- Swapping between multiple process implementations with differing number of processes.

The results of the single process implementations were conclusive. The deltas between the original implementations and the toolkit's swapped implementations were consistently zero. Swapping more complicated implementations involving multiple process implementations introduced some variability. The delta's for buffering strategy 1 were on the average of 1.41947E-07 seconds. Buffering strategy 2 performed worse, with an average delta of 1.25 seconds between observed and expected values. Lastly, swapping buffering strategy 1 into strategy 2's node model produced a delta's on average of 0.09 seconds between observed and expected values.

Table 5.10: OPNET-UNIT Test Results. This table shows the results of testing the "identity" process model, revealing that the framework behaves as expected.

Unit Testing Identity Model Results

Input:	Expected Output:	Pass:
Packet on streams (0-99)	forwarded	Y
Stat on streams (0-10)	forwarded	Y
Remote intrpt with ICI	forwarded ICI	Y

5.3 OPNET-Unit

5.3.1 Verification of Implementations. In order to verify the implementation, we install an "identity" process model for testing that follows the following rules:

1. Forward all packets on the same numbered output stream as the index of the input stream where a packet arrived.
2. Forward all remote interrupts, re-associating the associated ICI, to an auxiliary node.
3. Forward all statistic interrupts, preserving the statistic value, on the same index as the input stream where the statistic arrived.

5.3.2 A Motivating Example. Figure 5.5 shows a simple process model. This process model performs initialization procedures in the "INIT" state, and then transfers directly to the "WAIT" state where the process blocks until another interrupt is received. The INIT state initializes all state variables to 0. The WAIT state performs three simple arithmetic operations involving the incoming packet: calculating the largest, smallest and average packet size. The Code that executes in the "INIT" state is shown in listing 5.6.

The code that executes in the "WAIT" state when an interrupt is received is shown in listing 5.7.

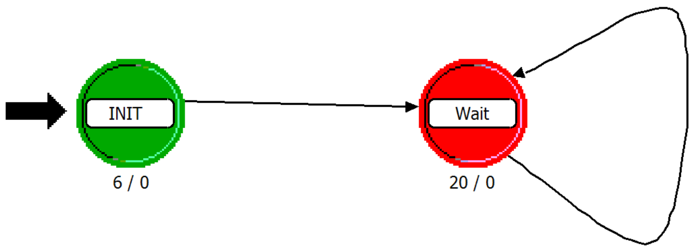

Figure 5.5: An Example Process Model Under Test. This process model contains only two states with an "INIT" state performing initializations, and the "Wait" state handling all subsequent interrupts.

```
size_t packet_max_size = 0;
size_t packet_min_size = 0;
size_t num_packets = 0;
size_t sum = 0;
double average = 0;
```

Listing 5.6: "INIT" State Proto-C. This listing shows the code contained in the "INIT" state of the process implementation under test. Line 2 contains a fault.

```
//recognize stream interrupts only
    if(op_intrpt_type() == OPC_INTRPT_STRM){
        //get the packet
        Packet* pkt = op_pk_get(op_intrpt_strm());
        //get the packet size
        packet_size = op_pk_bulk_size_get(pkt);

            //get the largest packet
            if(packet_size > packet_max_size);
                packet_max_size = packet_size;

             //get the smallest packet
             if(packet_size < packet_min_size);
                packet_min_size = packet_size;
        //calculate average
        num_packets++;
        sum += packet_size;
        average = sum/num_packets;
        op_stat_write(AvgSizeHandle,average);
        delete pkt;
}
```

Listing 5.7: . "Wait" State Proto-C. This listing shows the code contained in the "Wait" state of the process implementation under test. Lines 9 and 13 contain faults.

To test the process model, we write several tests to verify proper functioning. In accordance with good testing practices, we are particularly interested in boundary conditions. In this case, our boundary conditions, and thus test cases, are the following:

1. No packets sent to the node.
2. One packet sent to the node (size 0).
3. Two packets sent to the node (sizes 0 and INT_MAX).
4. Two packets sent to the node (sizes 1 and INT_MAX)

Using the CxxTest framework, we can implement the first non-trivial test case (Listing 5.8).

After running the tests, tests 3 and 4 fail. Utilizing the debugging concept of slicing discussed in [11], we narrow the bug search to lines 13 and 16 and discover empty control statements. We replace the statement:

```
void test2( void )
{
//get the first packet
Packet* pkt = harness->getPacket(0);
//send the packet to the node under test on stream 0
harness->sendPacket(pkt,0)
harness->incrementOneEvent();
TS_ASSERT_EQUALS(manager->getStateVariables()->packet_max_size,0);
TS_ASSERT_EQUALS(manager->getStateVariables()->packet_min_size,0);


TS_ASSERT_EQUALS(manager->getStateVariables()->num_packets,1);
TS_ASSERT_EQUALS(manager->getStateVariables()->sum,0);
TS_ASSERT_EQUALS(manager->getStateVariables()->average,0);
}
```

Listing 5.8: Non-Trivial Test Case. This shows the full implementation of the test case that ensures axiom 2 holds true.

```
(13) if(packet_size > packet_max_size);
(14)     packet_max_size = packet_size;
```

with:

```
(13) if(packet_size > packet_max_size)
(14)     packet_max_size = packet_size;
```

and similarly with the "if" statement in line 16. Retest. Now only test 4 fails. Again, slicing leads the debugging search directly to line 2 in the initialization state. We change:

```
(2) size_t packet_min_size = 0;
```

with:

```
(2) size_t packet_min_size = INT_MAX;
```

and retest. All tests now pass. Even in this small sample of code there were several common bugs, extra ";'s" and incorrect initializations, that could have been difficult to debug without an isolated testing environment in which to test axioms and re-create boundary conditions where program faults frequently lie.

5.3.3 Motivating Example 2. Figure 5.6 shows an actual process model provided by OPNET and demonstrates the second class of bug that can occur with

OPNET process models: faults in the mechanism of the process model execution (entering the wrong state at the wrong time). OPNET-Unit provides a mechanism to determine the state in which a particular process implementation resides. Through careful creation of input conditions, unit tests can verify that various paths through a given process model are traversed as expected.

5.4 Conclusion

This chapter performed numerous test scenarios on critical parts of the two major applications developed using the toolkit: OP-NPT and OPNET-Units. The OP-NPT tests verified that the externalized implementations are logically equivalent to the original process implementations provided by their original authors. The OPNET-Unit tests using the "identity" process implementation verify the correctness of the OPNET-Unit framework. Additionally, examples of defect localization utilizing the OPNET-Unit framework demonstrates its usefulness and overall utility.

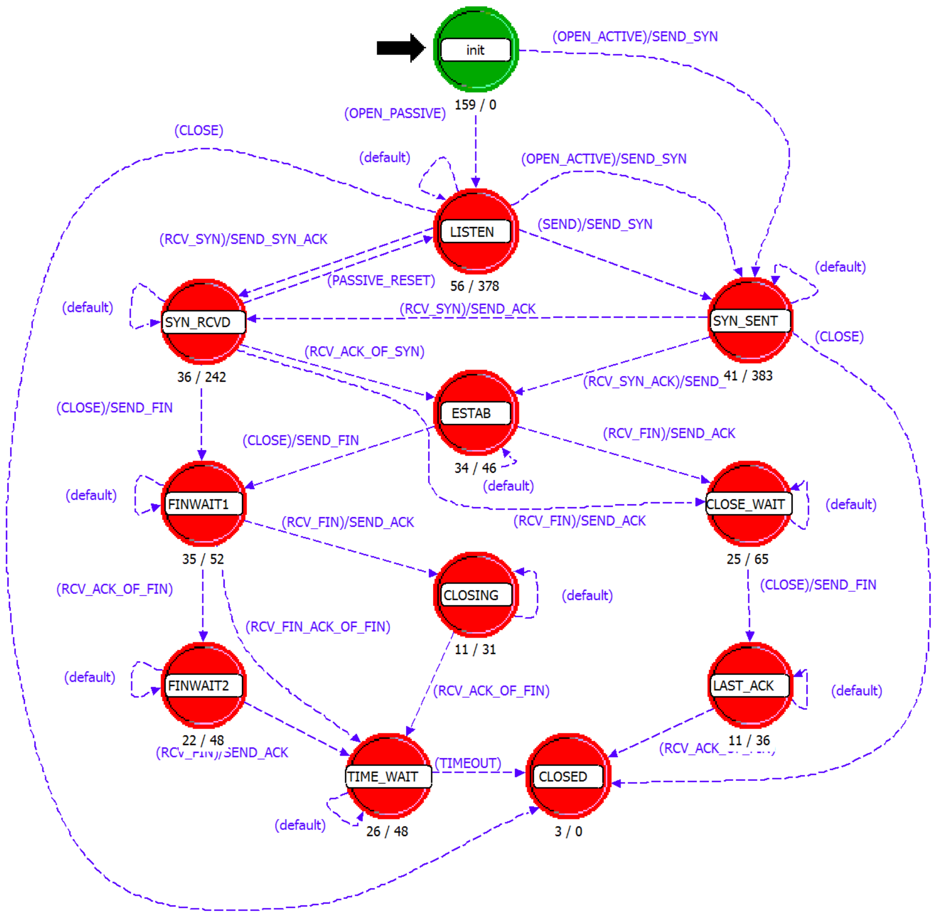

Figure 5.6: Complex Process Model. This actual process model that ships with OPNET demonstrates the second class of bug that can occur with OPNET process models: faults in the mechanism of the process model execution (entering the wrong state at the wrong time).

VI. Conclusion

The future possibilities created by building an IoC container around the OPNET Modeler environment are literally endless. By the very nature of dependency injection, the framework breaks the debilitating coupling between an OPNET simulation and the very implementations that define how the simulation functions. This thesis demonstrated two applications, only now possible because of the breaking of this coupling, OP-NPT and OPNET-Unit.

Moreover, these two applications are contributions in and of themselves. OP-NPT allows for the plug-and-play ability of OPNET implementations. Other domains have long since had this capability, but the proto-C environment coupled with the mechanisms OPNET uses to process the implementations prevents their plug-and-play ability. Moreover, once this mechanism was in place, the thesis verified that the plug-and-play implementations were logically equivalent to their original counterparts. Lastly, the thesis demonstrated the execution of a simulation with two previously unrelated algorithms integrated into a framework-enabled router.

OPNET-Unit also demonstrates several contributions. First, it demonstrates the ability to maintain intimate control over the execution of customized OPNET simulations through a well-defined interface. This allows native applications to utilize the full power of OPNET without maintaining coupling to a simulation. The utility of this mechanism was demonstrated through the design and implementation of OPNET-Unit. OPNET-Unit its self provides several advantages to the OPNET community. First, automated unit testing in general was not available to developers utilizing the OPNET paradigms. With the advent automated unit testing, developers can take advantage of other development methodologies already in use in the software engineering community such as Agile, XP, and TDD. These alternate development methodologies enable developers to spend more time developing, and less time debugging, thus producing an overall increase in code quality.

In summary the major (possibly publishable) and minor (non-publishable, but important to the Cyber ANiMAL Lab) are listed below:

Major contributions:

1. Transforming OPNET into an IoC container.
2. Developing an demonstrating "hot-swappabe" OPNET compatible protocol implementations and verifying their correctness.
3. Showing the result of the integration of two swappable implementations: strategic buffering mechanisms 1 and 2
4. Developing an OPNET process model unit testing framework that permits the automation of process implementation testing.

Minor contributions:

1. Developing an object-oriented error-handling OPNET scenario generation library.
2. Developing an API for precision execution of simulations.
3. Developing an API for "simulation-time" recovery of raw statistics from OPNET simulations.
4. Developing a JAVA based API for all libraries, useful for integration with JAVA based applications.
5. Developing an API for dynamic input from users about node and link status.

6.1 Future Recommendations

There is much potential future work with the dependency injection framework. The OP-NPT application could be expanded to handle other OPNET algorithms currently under development. Thus, a process implementation database could be developed and maintained, expanding the impact of this research. Moreover, the framework does not currently posses a mechanism for simulating dynamic topology related algorithms. The inclusion of algorithm's into the framework would require fundamentally new infrastructure in many places to facilitate the unique requirements of these types of algorithms. Additionally, the improvement of the integration of the framework with other research efforts, such as NetViz [3], would prove beneficial to the research community. Providing interfaces for not only displaying simulation related information in the visualization, but also for receiving feedback from the visualization about node and link status for “simulation time” update into the simulation. Lastly, the development of an abstract event queue with standardized interface would allow for the integration of non-OPNET based process implementations to be included in OPNET simulations. Developers working with NS-2 could, for example, provide NS-2 specific implementations of the generalized abstract event queue. This could potentially allow implementations that are simultaneously compatible with both OPNET, NS-2, and any other platform for which an implementation exists.

Bibliography

1. Autumn. http://code.google.com/p/autumnframework/, January 2008.
2. Bederson, Benjamin B., Jesse Grosjean, and Jon Meyer. "Toolkit Design for Interactive Structured Graphics". *IEEE Trans. Softw. Eng.*, 30(8):535–546, 2004. ISSN 0098-5589.
3. Beleue, J. Mark. "Network Visualization Design Using Prefuse Visualization Toolkit". Air Force Institute of Technology Thesis, March 2008.
4. Cooper, Ryan F. "Airborne Network and Datalink Technology Analysis Program: Distributed LInk 16 Simulation Demonstration". OPNETWORK 2007, 2007.
5. Dadarlat, C. Coffey T., V. Ivan. "A Middleware Based Approach for Designing Routing Protocols". *Electrical and Computer Engineering, 2002. IEEE CCECE 2002.*, 3:1436– 1441, 2002.
6. Dourish, Paul and W. Keith Edwards. "A Tale of Two Toolkits: Relating Infrastructure andUse in Flexible CSCW Toolkits". *Comput. Supported Coop. Work*, 9(1):33–51, 2000. ISSN 0925-9724.
7. Duggan, Dominic. "Type-based hot swapping of running modules (extended abstract)". *ICFP '01: Proceedings of the sixth ACM SIGPLAN international conference on Functional programming*, 62–73. ACM, New York, NY, USA, 2001. ISBN 1-58113-415-0.
8. Ford, Bryan, Godmar Back, Greg Benson, Jay Lepreau, Albert Lin, and Olin Shivers. "The Flux OSKit: a substrate for kernel and language research". *SOSP '97: Proceedings of the sixteenth ACM symposium on Operating systems principles*, 38–51. ACM Press, New York, NY, USA, 1997. ISBN 0-89791-916-5.
9. Fowler, Martin. *Patterns of Enterprise Application Architecture.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002. ISBN 0321127420.
10. Fowler, Martin. "Inversion of Control Containers and the Dependency Injection Pattern". http://martinfowler.com/articles/injection.html, January 2004.
11. Francel, Margaret Ann and Spencer Rugaber. "The Relationship of Slicing and Debugging to Program Understanding". *IWPC '99: Proceedings of the 7th International Workshop on Program Comprehension*, 106. IEEE Computer Society, Washington, DC, USA, 1999. ISBN 0-7695-0179-6.
12. Franz, Michael. "Dynamic Linking of Software Components". *Computer*, 30(3):74–81, 1997. ISSN 0018-9162.
13. Freeman, Steve, Tim Mackinnon, Nat Pryce, and Joe Walnes. "jMock: supporting responsibility-based design with mock objects". *OOPSLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems,*

languages, and applications, 4–5. ACM, New York, NY, USA, 2004. ISBN 1-58113-833-4.

14. Freeman, Steve, Tim Mackinnon, Nat Pryce, and Joe Walnes. "Mock roles, objects". *OOPSLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, 236–246. ACM, New York, NY, USA, 2004. ISBN 1-58113-833-4.

15. Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995. ISBN 0-201-63361-2.

16. Harmon, Duane. "Overcoming TCP Degradation in the Presence of Multiple Intermittent Link Failures Utilizing Intermediate Buffering". Air Force Institute of Technology Thesis, 2007.

17. Janzen, David S. and Hossein Saiedian. "A Leveled Examination of Test-Driven Development Acceptance". *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*, 719–722. IEEE Computer Society, Washington, DC, USA, 2007. ISBN 0-7695-2828-7.

18. Kohler, Eddie, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. "The click modular router". *ACM Trans. Comput. Syst.*, 18(3):263–297, 2000. ISSN 0734-2071.

19. Larman, Craig. *Applying UML and patterns: an introduction to object-oriented analysis and design.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1998. ISBN 0-13-748880-7.

20. Madhyastha, Harsha V., Arun Venkataramani, Arvind Krishnamurthy, and Thomas Anderson. "Oasis: an overlay-aware network stack". *SIGOPS Oper. Syst. Rev.*, 40(1):41–48, 2006. ISSN 0163-5980.

21. Meszaros, Gerard. *xUnit Test Patterns: Refactoring Test Code.* Addison-Wesley, 2007.

22. Mürk, Oleg and Jevgeni Kabanov. "Aranea: web framework construction and integration kit". *PPPJ '06: Proceedings of the 4th international symposium on Principles and practice of programming in Java*, 163–172. ACM, New York, NY, USA, 2006. ISBN 3-939352-05-5.

23. OPNET Technologies, 7255 Woodmont Avenue, Bethesda MD 20814-7904 USA. *Modeler Documentation Set*, 12.0 edition, 1987-2006.

24. PicoContainer. http://www.picocontainer.org/, January 2008.

25. Richards, Robert A. "An Intelligent Tool for Network Configuration and Optimization".

26. Schroeder, William J., Kenneth M. Martin, and William E. Lorensen. "The design and implementation of an object-oriented toolkit for 3D graphics and visualiza-

tion". *VIS '96: Proceedings of the 7th conference on Visualization '96*, 93–ff. IEEE Computer Society Press, Los Alamitos, CA, USA, 1996. ISBN 0-89791-864-9.

27. Seck Chai Lew, Ann Foo Cher Edwin, David Shiu Kei Cheung. "A Mix Executable Protocols and OPNET Simulation Environment For Rapid Protocols Development and Communication System Performance Evaluation". OPNETWORK 2007, 2007.

28. Spring. http://www.springframework.org/about, January 2008.

29. Stachtos, V., M. Kounavis, and A. Campbell. "Sphere: A Binding Model and Middleware for Routing Protocols", 2001. URL `citeseer.ist.psu.edu/stachtos01sphere.html`.

30. Tomcat. http://tomcat.apache.org/, January 2008.

31. Weinand, Andre, Erich Gamma, and Rudolf Marty. "ET++ an Object Oriented Application Framework in C++". *OOPSLA '88: Conference proceedings on Object-oriented programming systems, languages and applications*, 46–57. ACM Press, New York, NY, USA, 1988. ISBN 0-89791-284-5.

Vita

Graduating in 2006, 2d Lt Mark Coyne graduated from The Citadel in Charleston SC, and was commissioned into the Air Force. After applying for the Direct Accession program, Mark was accepted to AFIT. In March of 2008, he graduated with a Master's Degree in Computer Science, specializing in Software Engineering. Mark will next travel to Ft. Meade in Annapolis Maryland to work with the National Security Agency at the 70th Intelligence Support Squadron in support of the Air Force's emerging Cyber Operations career fields.

Permanent address: 2950 Hobson Way
Air Force Institute of Technology
Wright-Patterson AFB, OH 45433

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704–0188

1. REPORT DATE *(DD–MM–YYYY)* 27 March 2008	**2. REPORT TYPE** Master's Thesis	**3. DATES COVERED** *(From — To)* Aug 2006 — Mar 2008

4. TITLE AND SUBTITLE Hot Swapping Protocol Implementations in the OPNET Modeler Development Environment	**5a. CONTRACT NUMBER**
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S) Mark E. Coyne, 2d Lt, USAF	**5d. PROJECT NUMBER** ENG 08-175
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Air Force Institute of Technology Graduate School of Engineering and Management (AFIT/EN) 2950 Hobson Way WPAFB OH 45433-7765	**8. PERFORMING ORGANIZATION REPORT NUMBER** AFIT/GCS/ENG/08-05
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) Air Force Office of Scientific Research Robert J. Bonneau 875 N. Randolph Street Arlington, VA 22203-1768 (703) 696-6565 (DSN: 426-6207), email: david.luginbuhl@afosr.af.mil	**10. SPONSOR/MONITOR'S ACRONYM(S)**
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT

Approval for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

This research effort demonstrates hot swapping protocol implementations in OPNET via the building of a dependency injection testing framework. The thesis demonstrates the externalization (compiling as stand-alone code) of OPNET process models, and their inclusion into custom DLL's (Dynamically Linked Libraries). A framework then utilizes these process model DLL's, to specify, or "inject," process implementations post-compile time into an OPNET simulation. Two separate applications demonstrate this mechanism. The first application is a toolkit that allows for the testing of multiple routing related protocols in various combinations without code re-compilation or scenario re-generation. The toolkit produced similar results as the same simulation generated manually with OPNET. The second application demonstrates the viability of a unit testing mechanism for the externalized process models. The unit testing mechanism was demonstrated by integrating with CxxTest and executing xUnit style test suits.

15. SUBJECT TERMS

Discrete Event Simulation, Dependency Injection, Hot Swapping, Unit Testing

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**			Maj Scott Graham, PhD
U	U	U	UU	107	**19b. TELEPHONE NUMBER** *(include area code)* (937) 255-3636, x4918 scott.graham@afit.edu

Standard Form 298 (Rev. 8–98)
Prescribed by ANSI Std. Z39.18